Mastering Perl Prima

A Step-by-Step Guide for Beginners

Version 1.0
Reinier Maliepaard

Part 1 – Foundations

Part 2 – Core GUI Programming Concepts

Part 3 – Fundamental Widgets

13. **Selecting Widgets**
    - 13.1 Radio Buttons
    - 13.2 Checkboxes
    - 13.3 Combo Boxes
    - 13.4 List Boxes
    - 13.5 Checklists
    - 13.6 Draggable Items in Lists

Closing Words

Part 4 – Dialogs and User Interaction

14. **Standard and Custom Dialogs**
    - 14.1 Message Boxes (Info, Warning, Confirm)
    - 14.2 File Open/Save Dialogs
    - 14.3 Find and Replace Dialogs
    - 14.4 More Dialogs in Prima
    - 14.5 Modal vs. Modeless Dialogs
    - 14.6 Constructing Your Own Dialog

Closing Words

Extra: My Personal Message Utility

1. Why `wordWrap` Is the Best Solution for a Message Box That Fits the Text
2. Creating, invoking and modifying My Custom Message Window

Part 5 – Dynamic Widgets and Timers

15. **Time-Based and Status Widgets**

    - 15.1 Using the Time Widget and Timers
        - 15.1.1 Main Window Setup
        - 15.1.2 The Time Display (Widget::Time)
        - 15.1.3 Stopwatch State Machine
        - 15.1.4 Calculating and Updating the Display
        - 15.1.5 The Timer Object
        - 15.1.6 Start, Stop, Reset Buttons
        - 15.1.7 Full Program Stopwatch

    - 15.2 Progress Bars for Background Tasks
        - 15.2.1 Window and Time Display
        - 15.2.2 Timer Variables and Progress Bar
        - 15.2.3 Reset Function
        - 15.2.4 Timer Tick Logic (start_timer)
        - 15.2.5 Timer Object
        - 15.2.6 Duration Selector (ComboBox)
        - 15.2.7 Start/Stop Button
        - 15.2.8 Full Program Countdown Timer

    - 15.3 Toggle Buttons and Interactive State Controls
        - 15.3.1 The Main Window and Panel
        - 15.3.2 Status Label
        - 15.3.3 Toggle Button (Enabled Only in Mode C)
        - 15.3.4 Checkbox (Enabled Only in Mode B)
        - 15.3.5 Radio Modes and Dynamic State Switching
        - 15.3.6 Full Program Toggle Controls

Closing Words

Part 6 – Layout and Interface Organization

16. **Organizing Widgets Spatially**
    - 16.1 Using Containers and `pack`
    - 16.2 Static Multi-Pane Layouts
    - 16.3 Dynamic Panes with `FrameSet`
        - 16.3.1 `FrameSet`, `Label`, and Widget
        - 16.3.2 `FrameSet` and `ImageViewer`

17. **Structuring Your UI Logically**
    - 17.1 `GroupBox`: Visual Grouping
    - 17.2 `Notebook` – Tabs for Complex Interfaces
    - 17.3 `Panel`: Custom Borders and Backgrounds
        - 17.3.1 Example 1: A Simple Decorative Panel
        - 17.3.2 Example 2: Reusable Panels (Toolbar and Sidebar)
        - 17.3.3 Do You Really Need a Panel?
        - 17.3.3.1 A Status Bar
        - 17.3.3.2 A Status Window
        - 17.3.4 When to Use (or Skip) a Panel?

Closing Words

Part 7 – Advanced Customization

18. **Custom Widgets Through Composition and Inheritance**
    - 18.1 Understanding `ownerColor` and `ownerBackColor`
    - 18.2 Creating Reusable Widget Classes with Defaults

19. **Working with Table Widgets**
    - 19.1 Grid Widget Overview
    - 19.2 Basic Example
    - 19.3 Adding & Removing Rows and Columns
    - 19.4 Multi-Selection and Dynamic Updates

20. **DetailedList Widgets**
    - 20.1 Understanding `DetailedList`
    - 20.2 Building Your First DetailedList Interface
    - 20.3 Customizing Columns and Headers
        - 20.3.1 Customizing the Header
        - 20.3.2 Customizing the Items
    - 20.4 Sorting, Alignment, and Spacing
    - 20.5 Customizing Header
    - 20.6 Customizing Items
    - 20.7 headerClass property
    - 20.8 Populating and Rearranging Items from a File
    - 20.9 More on sorting

Closing Words

Part 8 – Extending Prima with Your Own Logic

21. **Strategies for Extending Prima Applications**
    - 21.1 Extension Strategy 1: Using Event Callbacks

Appendices

Part 1 - Foundations

Welcome to 'Mastering Perl Prima: A Step-by-Step Guide for Beginners'

(version 1.0)

1.1 Who This Tutorial Is For

This tutorial is for those familiar with Perl who want to explore graphical user interface (GUI) development using Perl Prima. If you already know Perl, this guide will help you leverage that knowledge to build interactive, user-friendly applications with Prima.

1.2 What You Should Already Know

To start with Prima, you should be comfortable with Perl basics:

- Variables and Data Types: using scalars (`$var`), arrays (`@array`), and hashes (`%hash`).
- Control Structures: writing loops (for, while) and conditional statements (if-else).
- Subroutines: defining and calling functions to organize your code.
- File Handling: reading from and writing to files.

If you're new to Perl, review these basics first. Once you're comfortable with these concepts, you'll be ready to build interactive GUI applications! Visit my website for additional tips, examples, and resources: https://reiniermaliepaard.nl/perl/

1.3 What You Will Learn

This tutorial covers essential concepts, including:

- Creating windows: learn how to initialize and configure your main application window.
- Managing widgets: discover how to work with various interface elements such as buttons, labels, and input fields.
- Handling events: understand how to respond to user interactions and system events effectively.
- Customizing behavior: explore how to personalize your application with custom actions and behaviors.

By the end, you'll have the skills to build and customize your own Perl Prima applications with interactive and responsive interfaces.

1.4 How to Use This Tutorial

You can use this tutorial in two ways:

On this website:

- Code examples can be copied and run directly without formatting issues.
- All images used in the examples can be downloaded All images used in the examples can be downloaded here.

- The code has been tested on Linux; visual results may differ between Linux distributions, themes, and operating systems such as Windows.
- This website also includes a search function, which make it easy to quickly find topics, commands, or examples.

PDF version:
A PDF version with an index is also available (download here), created by converting this website. Note: copying code from the PDF may break indentation, so the website is recommended when you want to copy and run the examples.

1.5 License

2. Why I wrote this tutorial on GUI Perl Prima

2.1 Making Perl Prima accessible

Perl Prima is a robust and versatile toolkit for building graphical user interfaces (GUIs) in Perl. Over the years, I've developed a variety of Windows applications - ranging from educational and math tools to music and data visualization software. Now that I've fully transitioned to Linux, I wanted to continue creating GUI tools, and Perl Prima has proven to be the ideal choice.

The official documentation, while comprehensive, can feel overwhelming for beginners because of its technical depth and specialized terminology. This tutorial aims to bridge that gap by presenting Prima in a clear, approachable way.

2.2 My journey and motivation

When I first started with Perl Prima, I was excited but quickly ran into challenges. The documentation, while thorough, often lacked a beginner-friendly approach with practical examples and clear explanations.

This inspired me to create a tutorial that breaks complex concepts into simple, actionable steps. My goal is to help you avoid the difficulties I encountered and make your learning journey smooth and enjoyable.

3. What Makes Prima Special

3.1 Cross-Platform Desktop GUI from One Codebase

Prima works seamlessly across Linux, Windows, and UNIX/X11 (including FreeBSD, IRIX, SunOS, Solaris, and more).

3.2 Modern User Interface Capabilities

Prima is a modern cross-platform GUI toolkit featuring a sleek, flat design that integrates well with contemporary desktop environments.

3.3 Visual Builder (RAD Tool)

Prima includes a RAD-style Visual Builder, a WYSIWYG tool for designing and testing GUIs quickly and intuitively.

3.4 Online Documentation and Learning Resources

- Online Documentation: Metacpan -Prima
- Documentation (PDF): Prima.pdf
- The package includes several useful examples, though some may be complex for beginners.

3.5 Maintenance and long-term support

Prima is actively maintained by its original developer, Dmitry Karasik, ensuring stability and long-term reliability.

4. Getting Started

4.1 Installing Prima

See *Appendix B* to learn how to install Perl Prima.

4.2 Your First Prima Script: Startup, Run Loop, Shutdown

Before exploring Prima in detail, let's create a small working program so you can see how a graphical application behaves. Let's make a compact version of a Prima application: a window (200x200) with a title and a centered button with text 'Exit!' and closes the application when the button is clicked. All in a few lines of code!

```
#!/usr/bin/perl
use strict;
use warnings;

use Prima qw(Application Buttons);

Prima::MainWindow->new(

    text => 'Your first Prima script',
    size => [200, 200]

)->insert(Button =>

    centered => 1,
    text => 'Exit!',
    onClick => sub { $::application->close }

);

Prima->run;
```

Listing 4.1: Example of a Compact Prima Application

Don't worry if the code seems complicated. Everything will become clear later on.

How to run it?

1. Save the code in a file named *`my_script.pl`*
2. Open a terminal or command prompt.
3. Run the script with Perl: *`perl my_script.pl`*

4.3 The Essential Boilerplate Explained

To save space, the following essential code will be omitted from the examples but should be included at the start of every script -see the complete listing files for reference:

```
#!/usr/bin/perl
use strict;
use warnings;
```

4.4 Naming Conventions

Although I don’t always follow the naming conventions in this tutorial, readers are encouraged to do so.
See *Appendix A: Naming Conventions* for the complete set of rules.

Key points to keep in mind:

- Variables: use descriptive names with camelCase (e.g., *`$mainWindow`*, *`$colorDialog`*).
- Widgets: prefix widget variables with the widget type (e.g., *`$btnSubmit`*, *`$lblStatus`*).

4.5 Clarity Over Brevity

While conciseness is often celebrated in programming, clarity should never be sacrificed for the sake of brevity. Redundancy, in this context, is not a flaw - it's a deliberate choice to make the code self-documenting and easy to understand at a glance. "Don't make me think" is a guiding principle here. Code should be immediately comprehensible, without mental parsing of clever shortcuts.

So I like:

```
push (@checked_options, $chk_lisp->text) if $chk_lisp->checked;
push (@checked_options, substr($chk_perl->text, 1)) if $chk_perl->checked;
push (@checked_options, $chk_python->text) if $chk_python->checked;
```

above

```
my @checked_options = map {
    $_->checked ? ( $_ == $chk_perl ? substr($_->text, 1) : $_->text ) : ()
} ($chk_lisp, $chk_perl, $chk_python);
```

5. Object-Oriented Programming (OOP) Refresher

In the previous chapter we created our first Prima program. You may have noticed syntax such as `Prima::MainWindow` and the `->` operator. These are part of Perl's object-oriented programming model. Because Prima is heavily object-oriented, it is useful to briefly review the core OOP concepts before we continue (how OOP is specifically used in Prima will not be covered here).

We'll first introduce general OOP concepts, and in later chapters, we'll show how they are implemented in Perl - the language on which Prima is built.

5.1 Classes and Objects

Prima is designed in an object-oriented style, which involves key concepts like classes, objects, properties, methods, and inheritance.

- **Class**: a blueprint that defines the structure and behavior of objects.
- **Object** (Instance): a concrete entity created from a class blueprint.

In short, a class defines what an object can do and contain, while an object is a real instance of that class.

This concept will be reflected in Perl using packages (classes) and bless (object creation). More on this in Section 5.5.

5.2 Properties and Methods

Objects are characterized by properties and methods:

- **Properties**: attributes or data that define the state of an object.
- **Methods**: functions or subroutines that manipulate the object's properties or perform actions.

In Perl, properties are usually stored in a hash reference, and methods are subroutines defined in the package representing the class.

5.3 Events and Event Handlers

In **event-driven programming**, applications respond to user or system actions (e.g., button clicks, mouse movements).

- **Event Handlers**: methods attached to objects that automatically execute when an event occurs.

Prima, being a GUI framework, heavily uses events, so understanding this concept helps when connecting Perl objects to interface actions.

5.4 Inheritance in Prima

Inheritance allows a class (subclass) to reuse attributes and methods from another class (superclass).

- This promotes code reuse and allows you to extend functionality without rewriting existing logic.
- In Perl, inheritance is implemented using the `@ISA` array or `use base`.

These concepts - classes, objects, methods, properties, and inheritance - will now be translated into Perl's OOP style, which is flexible and minimalistic.

5.5 OOP in Perl

Perl does not have a built-in `class` keyword like Java. Instead, it achieves OOP behavior using **packages** and **references**. Below is a concise guide to the key Perl OOP concepts.

5.5.1 Classes via `package`

- A Perl **class** is a **package** (namespace).
- Methods are subroutines defined inside the package.

```
package Animal;
package Dog;
```

5.5.2 Objects via `bless`

- Objects are typically **references** (often hash references) holding the object's data:

```
my $data = {};  # regular hash reference
```

- Use `bless` to associate the reference with a class:

```
my $object = bless($data, 'Dog');  # $object is now a Dog object
```

- Methods are called with the arrow operator:

```
$object->speak();  # calls Dog::speak if defined
```

`bless` does not alter the data; it simply tells Perl which package's methods to use.

5.5.3 Constructor (`new` method)

- Constructors are usually named `new`.
- They create and bless a data structure into the class.

```
my $d = Dog->new();
```

`new` is a regular method, not a keyword.

5.5.4 The `$self` variable

- Inside object methods, the first argument is the object itself.
- `$self` is the conventional name for this argument:

```
sub speak {
    my $self = shift;
    print "Dog says: Woof!\n";
}
```

5.5.5 Inheritance

- Perl supports single and multiple inheritance.
- To inherit from a parent class:

```
# Option 1: direct @ISA assignment
our @ISA = ('Animal');

# Option 2: use base
use base 'Animal';
```

- Subclasses inherit parent methods unless explicitly overridden:

```
package Dog;
use base 'Animal';

sub speak {
    my $self = shift;
    print "Dog says: Woof!\n";
}
```

5.5.6 An Example

```
# --- Base Class: Animal ---
{
    package Animal;
    sub new {
        my $class = shift;
        my $self = {};
        bless($self, $class);
        return $self;
    }
    sub speak {
        my $self = shift;
        print "Animal makes a sound\n";
    }
}

# --- Subclass: Dog ---
{
    package Dog;
    use base 'Animal';
    sub speak {
        my $self = shift;
        print "Dog says: Woof!\n";
    }
}

# --- Subclass: Cat ---
{
    package Cat;
    use base 'Animal';  # inherits speak
}

# --- Main Script ---
{
    my $a = Animal->new();
    my $d = Dog->new();
    my $c = Cat->new();

    $a->speak();  # Animal makes a sound
    $d->speak();  # Dog says: Woof!
    $c->speak();  # Animal makes a sound
}
```

To separate classes into files: place them in `Animal.pm`, `Dog.pm`, `Cat.pm` and load them in `main.pl` using `use Animal; use Dog; use Cat;`. Each `.pm` must end with `1;`.

5.5.7 Extending a Subclass

```
package Cat;
use base 'Animal';

sub new {
    my $class = shift;
    my $self = $class->SUPER::new();  # delegate to parent constructor
    $self->{color} = shift || 'black';
    return $self;
}

my $c = Cat->new('white');
print "Color cat: " . $c->{color} . "\n";  # Color cat: white
```

Using `SUPER::new()` avoids duplicating initialization logic and ensures a consistent object structure.

Closing words

GUI development in Perl Prima isn't just writing code; it's learning to put together windows, buttons, and event handlers so your program can interact with users. Take your time, try things out, and keep in mind that every application - no matter how refined - began with a simple first step.

Part 2: Core Concepts

6. Creating Main Window

Don't worry if this chapter feels abstract at first - we'll break it down step by step, and you'll see how it all fits together. In Part 1 you saw some basic OOP ideas in Perl. In this part, we will just use those ideas in practice; if any word feels unfamiliar, you can safely continue and return to the refresher later.

6.1 Anatomy of a `Prima::MainWindow`

Creating a main window is a fundamental step in building a graphical user interface (GUI) application. In Prima, this involves defining an instance of the `Prima::MainWindow` class.

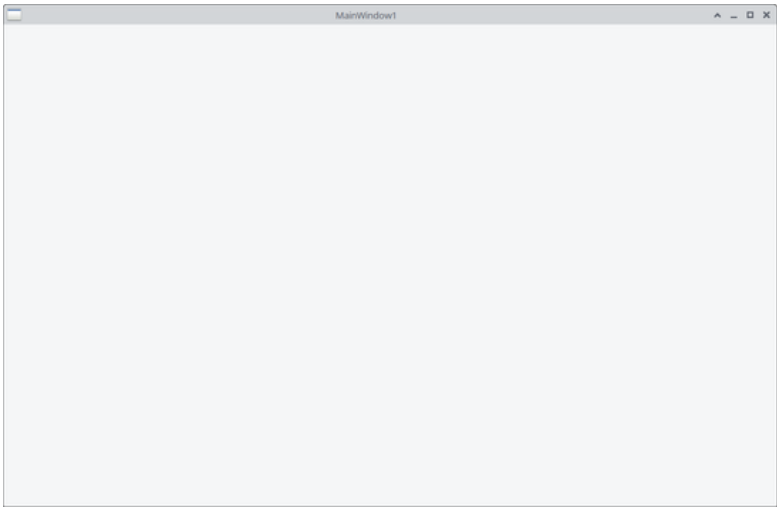


Figure 6.1: The Main Window

```
# load the modules Prima and Prima::Application
# Prima::Application sets up the application environment and controls
# the event loop
use Prima qw( Application );

# create a new window object, which is an instance of the  Prima::MainWindow
# class, and it will appear as a window rectangle on the screen

Prima::MainWindow->new(
    # here we define the window properties
);

# start the Prima event loop, which is required to run the GUI
Prima->run;
```

Listing 6.1: The Main Window

6.2 Essential Window Properties (Size, Title, Colors, Icon)

In this section, we define the window's basic appearance - its size, title, background, and icon - to establish a clear and consistent visual setup.

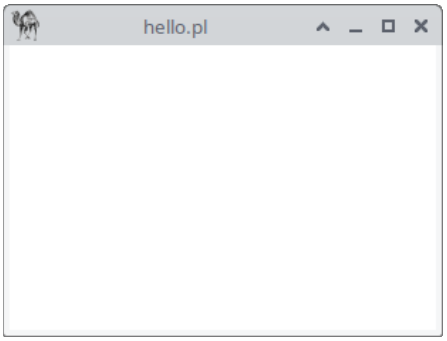


Figure 6.2: The Main Window with Configured Properties

```
use Prima qw(Application);

# A constructor is like a blueprint for creating objects. Here, new() is
# the constructor for Prima::MainWindow.

Prima::MainWindow->new(

    # the following are examples of 'Prima::MainWindow' class properties:
    # size, backColor, text, icon, and growMode. These will be set,
    # meaning they will be assigned values

    # set the window's size using a two-element array:
    # width = 300, height = 200
    size => [300, 200],

    # set the window's title text to the current script's filename hello.pl
    # $0 is the Perl special variable for the filename
    text => $0,

    # set the background color of the window to white (hexadecimal 0xFFFFFF
    # see https://www.color-hex.com/ and change # by 0x))
    backColor => 0xFFFFFF,

    # set the window's icon using the 'icon.png' file
    icon => Prima::Icon->load('icon.png'),

    # set the growMode property to position the window in the center of the
    # screen
    growMode => gm::Center,
);

Prima->run;
```

Listing 6.2: The Main Window with Configured Properties

A small note: what should you do if the icon file can't be found? A safer approach would be:

```
icon => (-e 'icon.png') ? Prima::Icon->load('icon.png') : undef,
```

However, Prima already handles missing icons internally, so it's usually nothing to worry about.

6.3 Storing and Accessing Window References

The main window (`MainWindow`) is created with the `new()` constructor of the `Prima::MainWindow` class. The resulting object is stored in the scalar variable `$mw`, which allows you to access the window, its properties, and its methods at any time simply through `$mw`.

```
use Prima qw(Application);

my $mw = Prima::MainWindow->new();
```

Note that that `$mw` is not *special*: it's just a variable.

6.4 Updating Properties Dynamically with `set()`

You can update properties dynamically with `set()`. This approach can be useful in certain scenarios, especially later in your program when you need to dynamically update properties.

```
use Prima qw(Application);

my $mw = Prima::MainWindow->new();

$mw->set(
    size => [300, 200],
    text => 'Hello',
    backColor => 0xFFFFFF,
);

Prima->run;
```

Listing 6.3: Setting the Main Window Properties Using the `set` Function

But it's often cleaner to define them upfront in `new()`. This helps avoid unnecessary screen updates and keeps your code organized. This approach helps prevent unnecessary resizing of the window and reduces initial screen flickering, leading to a smoother user experience. So, it is recommended to do this:

```
use Prima qw(Application);

my $mw = Prima::MainWindow->new(
    size => [300, 200],
    text => 'Hello',
    backColor => 0xFFFFFF,
);

Prima->run;
```

Listing 6.4: Setting the Main Window Properties Using the Constructor `new()`

6.4.1 Where does the 'set' method come from?

You might wonder, 'Where does the `set` method come from?' It's not listed in the `Prima::MainWindow` documentation because it's inherited from the `Prima::Widget` class. Think of `Prima::MainWindow` as a customized version of `Prima::Widget`. It gets all the tools from its parent, including `set()`, so you don't have to rebuild them from scratch. This is the power of inheritance in action!

6.4.2 Using direct accessors instead of `set`

In this tutorial, you'll often see direct property setters such as:

```
$label->text("hello");
```

This calls only the widget's `text()` setter, without the additional overhead of `set()`. Because it touches just a single property, it avoids unnecessary layout recalculations and helps reduce the risk of flickering, especially when updating text frequently.

7. Understanding Widgets

7.1 What Widgets Are

Widgets (in other languages called components or controls) are the building blocks of your GUI. Think of them as interactive tools - like buttons, text boxes, or checkboxes - that let users interact with your application. In Figure 7.1 the application has one user-interface (UI) widgets: a Button widget that serves as the Exit button.

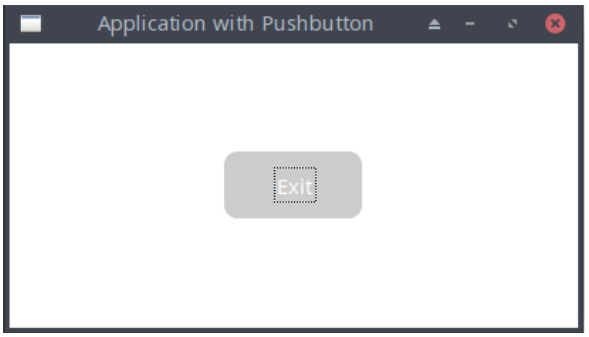


Figure 7.1: The Main Window with a Button Widget

In Figure 7.2, the application has three user-interface (UI) widgets: a label widget (displaying the text ‘Which programming language(s) do you know?’), six checkbox widgets and a button widget that serves as the Evaluate button.

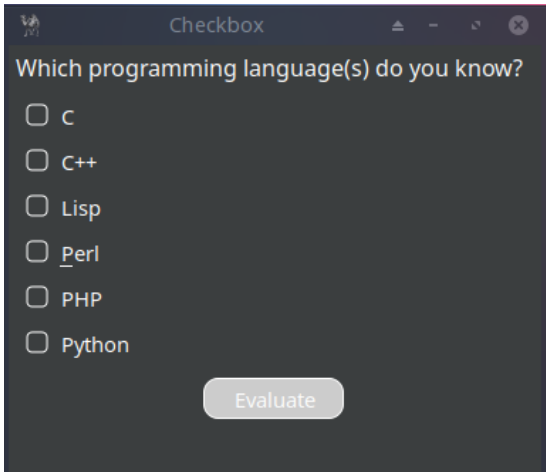


Figure 7.2: The Main Window with a Label Widget, Six Checkbox Widgets, and a Button Widget

Notice: the other graphical elements of the window shown in Figure 7.1 and 7.2, such as the title bar and the control buttons on the left and right sides of the title bar, are not Prima widgets. Rather, they are created and managed by the underlying graphical system in use. Prima only manages widgets; the desktop environment manages window frame elements.

7.2 Overview of Core Widget Categories

Widgets in Prima are organized like a family tree. At the top is the `Prima::Widget` class, which provides basic features. Other widgets, like buttons or labels, inherit these features and add their own specialties. This makes it easy to customize and reuse code.

Widgets can serve different roles:

- Function as windows or interface controls
- Contain other widgets (container widgets)
- Process various events (clicks, keyboard, mouse, timers)

Widgets come in three main categories:

1. **Basic Widgets**
    - Button: standard clickable button
    - Label: displays static text
    - CheckBox: for boolean selections
    - Radio: single-choice selection within a group
    - ComboBox: combined text input and dropdown
    - ListBox: selectable item container
    - InputLine: single-line text input

- Edit: multi-line text editor

2. **Interactive Widgets**
   - Menu: traditional dropdown menu bar
   - Dialog: base for dialog windows

3. **Container Widgets**
   - Widget: fundamental base class
   - Window: main application container
   - GroupBox: logical grouping container (especially for radio buttons)
   - Notebook: tabbed interface organizer

7.3 Common Properties, Methods, and Events

Every widget in Prima has properties and methods that define how it looks and behaves. Properties are like settings - such as color, size, or text - while methods are actions you can perform, like showing or hiding a widget. Notice that for the methods of widgets like MainWindow, Buttons, Labels etc., you should first know the properties and methods of their parent! It means that you should know the features of `Prima::Widget`

Let's explore some of the most useful properties and methods, ideal for initial usage and essential configurations (look further at https://metacpan.org/dist/Prima/view/pod/Prima/Widget.pod under API).

1. **Properties** define how a widget looks and behaves:
   - **name**: the widget's identifier.
   - **text**: the label or content displayed.
   - **color**: foreground color *)
   - **backColor**: background color
   - **visible**: boolean (1 = visible, 0 = hidden).
   - **geometry**: size (width/height) and position (X/Y coordinates).
   - **growMode => gm::Center:** positions the window in the center of the screen;

2. **Methods** let you interact with the widget programmatically:
   - **show()**: makes the widget visible.
   - **hide()**: hides the widget.
   - **update()**: refreshes the widget's content.
   - **focus()**: sets input focus to the widget
   - **close()**: call can_close() and if succesful, destroys the widget
   - **destroy()**: destroys the widget...brute force! Immediately removes the widget from memory. Use this method with caution, as it bypasses safety checks (like close(). It's best for situations where you need to forcefully remove a widget.

**) Colors can be defined using hexadecimal values (e.g., `0xFFFFFF` for white) or easy-to-remember constants like `cl::Red` or `cl::Blue`. For a full list of color options, check out (see https://metacpan.org/pod/Prima::Const under cl::- colors). Don't be afraid to experiment with colors to make your application visually appealing!*

Properties and methods are your tools for customizing widgets. Try changing the `backColor` of a button or using `show()` and `hide()` to toggle visibility. Experimenting is the best way to learn!

Before we add a widget to our MainWindow, let's take a moment to understand how event programming works - the key to making our interface interactive. Once we know how events connect user actions to program responses, we'll be ready to insert a button widget that actually *does* something.

8. Event-Driven Programming

Event-driven programming is all about responding to user actions. Imagine your application as a smart home: when someone rings the doorbell (an event), the system automatically turns on the lights (a response). In Prima, you define these responses using event handlers.

Event-driven programming might sound complex, but Prima makes it straightforward. When your program runs:

1. Prima automatically starts an event loop that continuously monitors for user interactions
2. Your widgets (buttons, input fields, etc.) generate events when users interact with them
3. You simply define what should happen for each type of event

8.1 The Event Loop and Application Flow

The event loop is the core of Prima's operation:

- It constantly checks for new events (clicks, keypresses, etc.)
- When an event occurs, it finds and executes the matching event handler
- After processing, it immediately returns to monitoring for new events

The event loop runs automatically in the background. It constantly listens for user actions - like clicks or keypresses - and triggers the appropriate response. You don't need to manage it manually; Prima handles it for you!

8.2 Handling User Input

Prima handles multiple simultaneous events by:

- Placing events in a queue
- Processing them in the order they occurred
- Ensuring your application remains responsive

8.3 Event Handlers and Callbacks

Event handlers define how your program reacts to user actions. Each handler responds to a specific event—for example, a mouse click or a key press. In Prima, the names of these handlers always start with ***on*** followed by the event name, such as `onClick`, `onKeyDown`, or `onMouseMove`. This makes it easy to see which event each handler belongs to.

Your task as the programmer is simple:

- **For basic actions:** use the built-in event handlers
- **For custom behavior:** attach your own callback functions

A *callback* is just a regular Perl function that Prima calls automatically when the event occurs. Here is a basic button click handler:

```
onClick => sub {
    # Close the application when the button is clicked
    $::application->close;
}
```

You can also use **named callbacks**, which helps keep your code readable as handlers become more complex:

```
onClick => \&do_exit,

sub do_exit {
    $::application->close;
}
```

The event-driven model means that:

- You never need to write complex event-management code
- Logic stays separate from the user interface
- Event sequencing is handled automatically

As a result, you can focus on *what* should happen, and Prima takes care of *when* it happens.

8.4 Event Handling and the special @_

Event handling refers to the mechanism by which the program responds to various events, such as user actions (like clicking a button or entering text) or system-generated events (like timers or window resizing). Event-driven programming is especially common in graphical user interfaces (GUIs) and server applications.

8.4.1 How Event Handling Works

1. Events: these are actions or occurrences, such as clicking a button, typing in a text field, or resizing a window.
2. Event Handlers: these are subroutines or callbacks that are executed in response to an event. When an event occurs, the associated event handler is triggered to handle that event.
3. Binding Events: in GUI toolkits like Prima, you can bind specific events (like `onClick` or `onChange`) to event handlers for different widgets (buttons, text fields, etc.).

8.4.2 The special array @_

In Perl and Prima, subroutines, including event handlers, receive their arguments via the special array @_. This array contains all the parameters passed to the subroutine. For event handling, @_ typically holds:

- `$_[0]`: the widget (or object) that triggered the event. This is often referred to as `$self` in object-oriented Perl because it refers to the current object.
- `$_[1]`, `$_[2]`, etc.: these may contain additional information about the event, such as details specific to the type of event (e.g., a key press or mouse click).

Using `@_` allows event handlers to respond dynamically based on which object triggered the event and other relevant details.

8.4.3 Example

In this tutorial you'll find many examples of the special array @_. For now an easy example. If you have multiple buttons and want to identify which one was clicked, you can use `$_[0]` or `$self` to reference the specific button that triggered the event: which button was clicked?

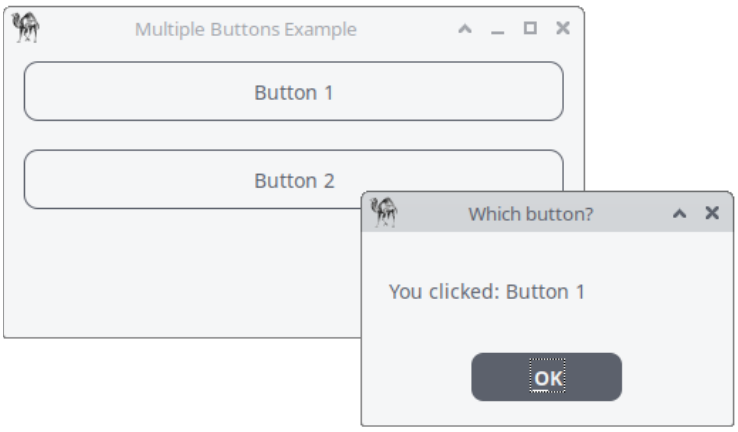


Figure 8.1: Which Button Was Clicked?

```
use Prima qw(Application Buttons Label MsgBox);

my $mw = Prima::MainWindow->new(
    text => 'Multiple Buttons Example',
    size => [400, 200],
    icon => Prima::Icon->load('icon.png'),
);

$mw->insert( Button =>

    pack   => {fill => 'x', side => 'top', pad => 20},
    text   => 'Button 1',
    onClick => sub {
                        my($self) = @_;
                        message_box("Which button?",
                                    "You clicked: " . $self->text,
                                    mb::OK,
                                    compact => 1 );
    },
);

$mw->insert( Button =>

   pack   => {fill => 'x', side => 'top', pad => 20},
   text   => 'Button 2',
   onClick => sub {
                       message_box("Which button?",
                       "You clicked: " . $_[0]->text,
                       mb::OK,
                       compact => 1 );
   },
);

Prima->run;
```

Listing 8.1: Which Button Was Clicked?

8.4.4 Some events and @_

A button widget (e.g., *`Prima::Button`*) has several event handlers that can be used to capture different types of user interactions. Below common events :

1. **Click** – fired when the button is clicked:

```
my ($self) = @_;
onClick => sub { ... };
```

2. **MouseDown / MouseUp** – fired when the mouse button is pressed or released:

```
my ($self, $btn, $mod, $x, $y) = @_;
onMouseDown => sub { ... };
onMouseUp   => sub { ... };
```

3. **MouseMove** – triggered when the mouse moves over the button:

```
my ($self, $mod, $x, $y) = @_;
onMouseMove => sub { ... };
```

4. **KeyDown / KeyUp** – fired when a key is pressed or released while the button has focus:

```
my ($self, $code, $key, $mod[, $repeat]) = @_;
onKeyDown => sub { ... };
onKeyUp   => sub { ... };
```

5. **MouseEnter / MouseLeave** – fired when the pointer enters or leaves the button area:

```
my ($self, $mod, $x, $y) = @_;
onMouseEnter => sub { ... };

my ($self) = @_;
onMouseLeave => sub { ... };
```

Later we'll explore how to discover which events a widget supports.

8.4.5 Application keyboard shortcuts, bind to a button

To bind a keyboard shortcut like `CTRL+M` to a button, you would typically use the `onKeyDown` event.

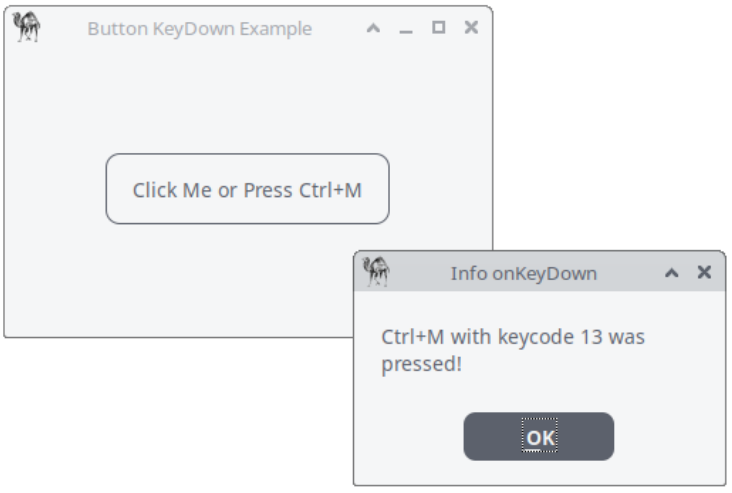


Figure 8.2: onKeydown Event

```
use Prima qw(Application Buttons Label MsgBox);

my $mw = Prima::MainWindow->new(
    text => 'Button KeyDown Example',
    size => [335, 200],
    icon => Prima::Icon->load('icon.png'),
);

# button with an onClick event (for mouse clicks)*
# and onKeyDown event (for keyboard events)*

my $button = $mw->insert(Button =>
    text   => 'Click Me or Press Ctrl+M',
    origin => [0, 80],
    size   => [200, 50],
    growMode => gm::Center,
    onClick => sub {
        message_box("Info onClick",
                    "Button was clicked!",
                    mb::Ok,
                    compact=>1, );
    },
    onKeyDown => sub {
        # event handler receive their arguments via the special array @_
        my ($self, $keycode, $key, $mod, $repeat) = @_;

        # check if Ctrl is pressed and key is Ctrl+M (which is keycode 13)
        # keycodes can differ depending on platform; Ctrl+M on my system is
        # ASCII 13.
        if ($mod & km::Ctrl and $keycode == 13) {
            message_box("Info onKeyDown",
                        "Ctrl+M with keycode " . $keycode . " was pressed!",
                        mb::Ok, compact=>1,);
        }
    },
);

# set focus to the button when the application starts, so it can capture keyboard events
$button->focused(1);

Prima->run;
```

Listing 8.2:: onKeydown Event

Closing Words

You now have all the foundational concepts needed to understand how Prima applications are built and how they behave. In this part, we explored how to create windows, how widgets are structured, and how properties and methods shape their appearance and behavior. Most importantly, you learned how event-driven programming allows your application to react to user actions - and how Prima uses the special array `@_` to pass essential information to your event handlers.

These ideas may feel new at first, but with practice they quickly become natural. As you continue, you'll discover how these basic tools combine to build complete, interactive interfaces.

Your journey with Prima is just beginning - let's continue!

Part 3 - Basic Widgets

9. Buttons and User Actions

9.1 Adding Buttons to the MainWindow

An essential control element of the MainWindow object is the button. This is a familiar component in many windows - think of the buttons you see in dialog boxes, typically labeled "OK" or "Cancel." While buttons can also display images, we'll focus on the more common use case for now: a button with simple text.

In Chapter 6, we created the MainWindow, which functions as a widget itself. The MainWindow organizes and contains various elements, such as Buttons, Labels, Input Fields, List Boxes, and more. Essentially, the MainWindow acts as the parent widget, managing all other components within the application, with those internal widgets serving as its children (refer to Figures 2 and 3 of section 7.1).

Keep in mind that since the MainWindow is a widget, adding another widget like a Button means they share similar properties. Therefore, the table in the previous chapter is applicable. Let’s explore how this works.

9.2 Creating Reusable Exit Buttons

We'll create a window with dimensions of 400 x 200, a title, a white background, centered on the screen, and an icon. You should already know how to do this. The widget we want to insert is a Button, sized 100 x 50, with text, a grey background, centered, and an action that triggers when clicked.

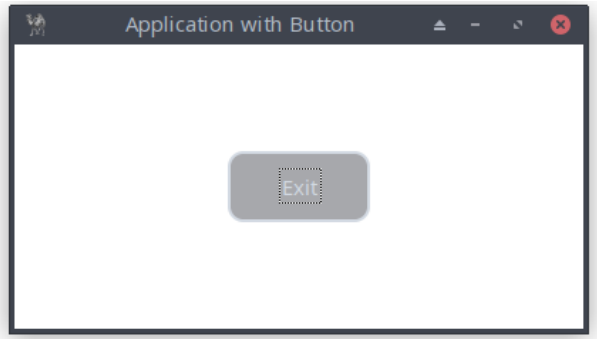


Figure 9.1: The Main Window with a Button Widget

Here’s the code:

```
# invoke the modules Application, Buttons
use Prima qw(Application Buttons);

# the following should be clear to you
my $mw = Prima::MainWindow->new(
    size => [400, 200],
    text => 'Application with Button',
    backColor => 0xFFFFFF,
    growMode => gm::Center,
    icon => Prima::Icon->load('icon.png'),
);

$mw->insert( Button =>

    # both the Button and MainWindow widgets share common properties
    size => [100, 50],
    text => 'Exit',
    backColor => 0xcccccc,
    growMode => gm::Center,

    # the `onClick` event is linked to a subroutine that handles closing
    # the window
    onClick => sub { $::application->close }
);

Prima->run;
```

Listing 9.1: The Main Window with a Button Widget

Let's explore how the *onClick* event works

- The most common event for buttons is *onClick*
- The *onClick* event here is called when the user presses (or otherwise activates) the button.
- *$::application* is a special global variable in Prima that refers to the main application object. It represents the top-level application instance that manages all windows and widgets.
- *->close*: this method is called on the application object to terminate the application. When you invoke *$::application->close*, it triggers the closing process for the entire Prima application, which effectively closes all open windows and ends the application's event loop.

Of course, the subroutine can be extended, which we explore in the next section.

9.3 Default Buttons and Tooltips

Now let's add two nice button features: default and hint

9.3.1 Setting the Default Button Property

Property	default
Description	Determines whether the button responds when the user presses the **Enter** key. When enabled (value: 1), the button is visually distinguished with a black border, indicating that it

	performs the default action (e.g., confirming a dialog or submitting a form).
Default Value	0 (disabled)

Table 9.1: Button Property default

```
$mw->insert( Button =>

    # both the Button and MainWindow widgets share common properties

    size => [100, 50],
    text => 'Exit',
    backColor => 0xcccccc,
    growMode => gm::Center,

    # marks the button as the default button, activated by Enter
    default => 1,

    onClick => sub { $::application->close }
);
```

9.3.2 Adding a Tooltip with the Hint Property

Property	hint
Description	A text string is shown under the mouse pointer if it is hovered over the widget.
Example	`hint => 'Press Enter to activate the button',`

Table 9.2: Tooltip with the Hint Property

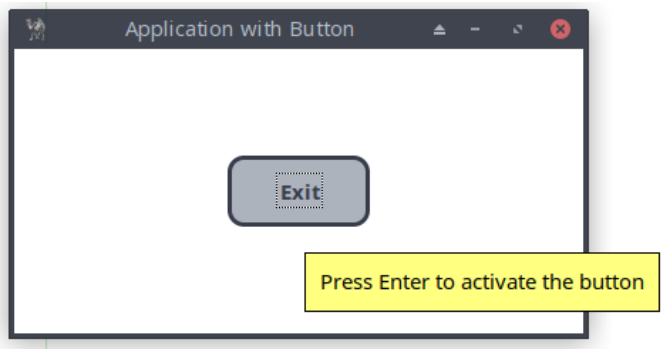


Figure 9.2: The Main Window with a Button that responds to the Enter Key and Includes a Tooltip.

9.4 Showing Alerts and Messages from Events

In the previous section, we created the MainWindow with a Button widget, where the `onClick` method (or better 'event') directly closes the window. To confirm the user's intent, a message box is useful. Prima offers a default message box (we'll create our own later).

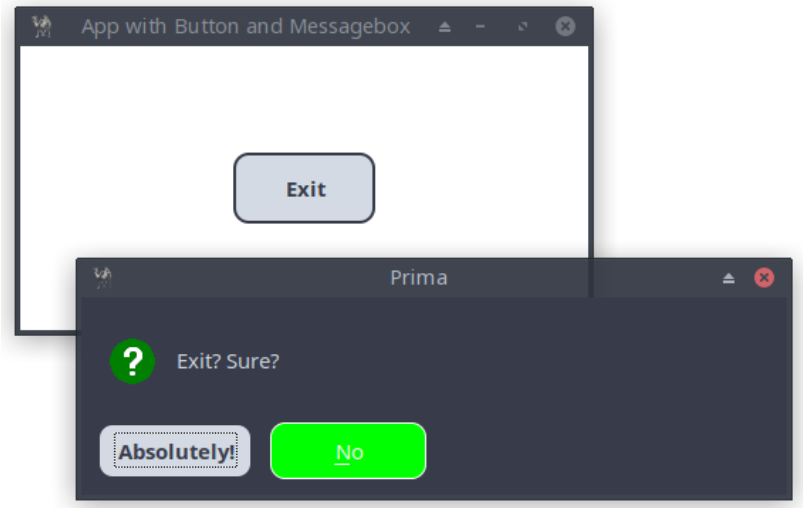


Figure 9.3: The Main Window with a Button, Displaying a Confirmation Messagebox

```
# Invoke the modules Application, Buttons, MsgBox
use Prima qw(Application Buttons MsgBox);

my $mw = Prima::MainWindow->new(
    size => [400, 200],
    text => 'App with Button and Messagebox',
    backColor => 0xFFFFFF,
    growMode => gm::Center,
    icon => Prima::Icon->load('icon.png'),
);

$mw->insert( Button =>
    size => [100, 50],
    text => 'Exit',
    backColor => 0xcccccc,
    default => 1,
    growMode => gm::Center,

    # The `onClick` event is now connected to the subroutine showMessage that
    # displays the default Prima message box before closing the window.
    # The reference variable $clicked_button_id stores a number associated with
    # the clicked button (let's say 'id'): mb::Yes corresponds with 2 and
    # mb::No with 8

    onClick => sub {
                        my $clicked_button_id = showMessage();
                        $::application->close if ($clicked_button_id == 2);
                    }
);

# subroutine myMessage
sub showMessage {

    # make a messagebox with two buttons Yes and No and a Question mark
    my $clicked_button_id = Prima::MsgBox::message('Exit? Sure?',
        mb::Yes|mb::No|mb::Question,
        buttons => {
                            mb::Yes => {
                             #text of the button that overrides the default 'Yes'
                             text => 'Absolutely!',
                            },

                            mb::No => {
                                        backColor => cl::LightGreen,
                                        color => 0xFFFFFF,
                            },
                    },
        );
}

Prima->run;
```

Listing 9.2: The Main Window with a Button, Displaying a Confirmation Messagebox

Consider adding the following two properties to showMessage():

```
# if 1, the message box width is shrunken so that there's no empty space
# on to the right.
compact => 1,

# defines the default button in the dialog
defButton => mb::No,
```

For now, this is enough to know about the button widget and its properties, methods, and events.

10. Displaying Text: Labels

10.1 Labels, Captions, and Text Elements

We will now create a simple window application that generates a random number when a button is clicked. The MainWindow contains two widgets: the familiar Button and the new Label. A Label widget is a simple widget used to display static text. It is often used to provide descriptions, titles, or any non-interactive text information to users within the application. Here the *onClick* event is defined as:

```
onClick => sub {
                    my $random_number = int(rand(100));
                    $label->set(text => $random_number);
               },
```

The reference variable *$label* calls the *set* function to assign the random number to the text property.

The basic setup of the MainWindow is extended by including the module Label

```
use Prima qw(Application Buttons Label)
```

The *origin* property is set in the Button and Label definitions. This property controls the position of the two widgets, with [0,0] representing the x-coordinate 0 and y-coordinate 0, which corresponds to the lower-left corner of the window. The button is positioned at [50, 75], and the label above it, is located at [110, 125].

Lastly, note that the setup of the MainWindow introduces new properties:

```
borderStyle => bs::Dialog
# preventing it from being resized;
# we've also reduced the number of border icons using the property
borderIcons => bi::SystemMenu|bi::TitleBar
```

Figure 10.1: The Non-resizable Main Window with a Button and Label widget

```
# invoke the modules Application, Buttons, Label
use Prima qw(Application Buttons Label);

my $mw = Prima::MainWindow->new(

    text => 'Random',
    size => [250, 200],

    # disable resize of the window:
    borderStyle => bs::Dialog,

    # show system menu button and/or close button (usually with icon) and
    # show also the title bar ('bi::TitleBar' is needed due to borderStyle)
    borderIcons => bi::SystemMenu|bi::TitleBar,
);

my $label = $mw->insert( Label =>
    # position widget: [x, y]
    origin => [110, 125],
    size  => [50, 50],
    text  => '',
    font  => {size => 26},
);

$mw->insert( Button =>
    # position widget: [x, y]
    origin => [50, 75],
    size   => [150, 50],
    text   => 'Random number!',
    default => 1,
    onClick => sub {
                        my $random_number = int(rand(100));
                        $label->set(text => $random_number);
                },
);

Prima->run;
```

Listing 10.1: The Non-resizable Main Window with a Button and Label widget

10.2 Auto-Resizing and Dynamic Layout

The Label widget has some common used properties:

a. Horizontal Alignment (***alignment***)

This property determines how text is aligned horizontally within the widget. You can set it to one of the following constants:

- `ta::Left` (default)
- `ta::Center`
- `ta::Right`

For example, use `ta::Center` to center-align text in a button or label.

b. Automatic Height Adjustment (`autoHeight`)

When enabled (`1`), this property allows the widget to **automatically adjust its height** to accommodate changes in the text content. If disabled (`0`, default), the widget retains a fixed height regardless of text length.

c. Automatic Width Adjustment (`autoWidth`)

When enabled (`1`, default), the widget **automatically adjusts its width** to fit the text content. If disabled (`0`), the widget maintains a fixed width, and text may be clipped or truncated if it exceeds the available space.

d. Text Wrapping (`wordWrap`)

This property controls how text wraps within the widget:

- If disabled (`0`, default), text only wraps at explicit newline characters.
- If enabled (`1`), text wraps automatically to fit the widget's width, ensuring all content remains visible.

e. Vertical Alignment (`valignment`)

This property sets the vertical alignment of text within the widget. You can choose from:

- `ta::Top` (default)
- `ta::Middle`
- `ta::Bottom`

For example, use `ta::Middle` to vertically center text in a label or button.

For a quick overview:

Property	Meaning	Default
`alignment`	Horizontal text alignment	`ta::Left`
`valignment`	Vertical text alignment	`ta::Top`
`autoHeight`	Adjust height to text	`0`
`autoWidth`	Adjust width to text	`1`
`wordWrap`	Text wraps automatically to fit the widget's width	`0`

Table 10.1: Main Properties

The following program demonstrates the power and flexibility of Prima's Label widget, specifically focusing on the `autoHeight` property. This property allows the label to automatically adjust its height based on the amount of text it contains, making it adaptable for dynamic content. This feature is particularly useful when handling varying or multiline text that may change based on user interactions or external data updates.

The Label widget in this program is initialized with specific properties:

- autoHeight set to 1, enabling the label to adjust its height automatically based on the text content (the default value is 0),
- an initial text of "line1" with a blue background and a visible border, placed at coordinates [50, 85] with a width of 300 pixels.

The Button widget titled "Add text" is included to simulate content changes. When clicked, it updates the text property to include multiple lines, triggering the autoHeight feature. As a result, the label will expand vertically to fit all lines without manually resizing it.

Notice the use of this command, which defines the width and height variables from `$mw->size`:

```
my ($mw_width, $mw_height) = $mw->size;
```

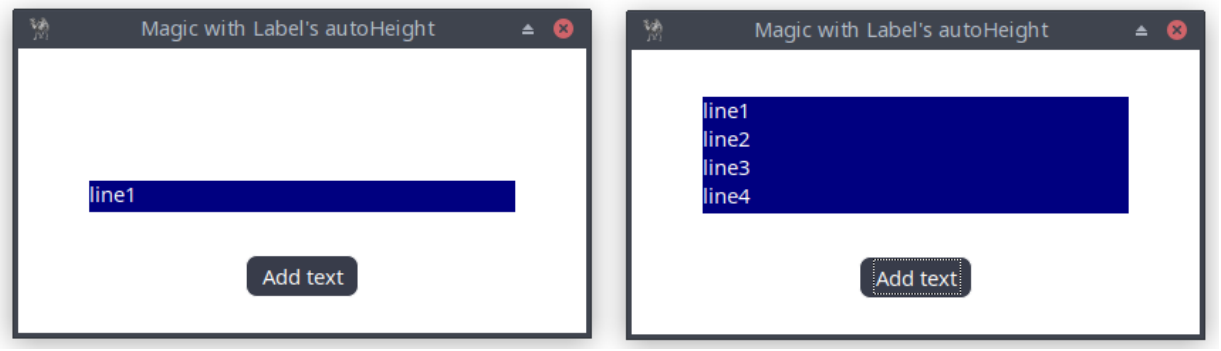


Figure 10.2: The Main Window with a Button and a Label Widget, Which Automatically Adjusts Its Height

```
use Prima qw(Application Label Buttons);

my $mw = Prima::MainWindow->new(

    text => 'Magic with Label's autoHeight',
    size => [400, 200],
    backColor => 0xFFFFFF,
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
    icon => Prima::Icon->load('icon.png'),
);

my ($mw_width, $mw_height) = $mw->size;

my $label = $mw->insert( Label =>

    origin => [50,85],
    size => [($mw_width-100), 30],
    text => 'line1',
    autoHeight => 1,
    color => 0xFFFFFF,
    backColor => cl::Blue,
);

$mw->insert( Button =>

    # center the button
    origin => [($mw_width/2)-(80/2), 25],
    size => [80, 30],
    text => 'Add text',
    color => 0xFFFFFF,
    onClick => sub { $label->text("line1\nline2\nline3\nline4"); },
);

Prima->run;
```

Listing 10.2: The Main Window with a Button and a Label Widget, Which Automatically Adjusts Its Height

10.3. Using a Panel to Manage Text Alignment and Word Wrapping in Multi-Widget Windows

Now we build a "Tip of the Day" application. It features a main window with a custom icon, a quote display, and buttons for navigating between quotes and exiting the application.

A `Panel` widget provides a bordered, light-gray background that visually organizes the main content area, enhancing readability.

```
# Add a panel to act as a border around the label

my $panel = $mw->insert( 'Widget::Panel' =>

    origin => [20, 60],
    size => [300, 120],
    width => 300,
    height => 120,
    ownerBackColor => 0,
    backColor => cl::LightGray,
    raise => 0,
    borderWidth => 1,
);
```

Inside this `Panel`, a `Label` widget - displaying the quotes - serves as a child element, inheriting its layout context. We add the properties alignment and wordWrap.

```
$panel = $mw->insert( Label =>

    origin => [25, 70],
    size => [285, 100],
    text => $quotes{$index},
    alignment => ta::Left,
    font => { size => 8, },
    wordWrap => 1,
    backColor => cl::LightGray,
    color => cl::Black,
);
```

Users can cycle through quotes stored in a `%quotes` hash by clicking the "Next tip" button, which updates the text in the `Label`.

```
onClick => sub {
                    ($index < (scalar keys %quotes)) ? $index++
                                                     : $index = 1;
                    $panel->text($quotes{$index});
                },
```

A custom information icon was loaded using the following code:

```
# Load a custom information icon

my $logo = Prima::Icon->load('tip.png');

# Insert the icon into the main window

$mw->insert( ImageViewer =>

    image => $logo,
    origin => [20, 200],
    size => [32, 32],
);
```

Figure 10.3: Tip of the Day Application

The complete code:

```
use Prima qw(Application Label Buttons ImageViewer Widget::Panel);

my $mw = Prima::MainWindow->new(

    text => 'Tip of the Day',
    size => [350, 250],
    backColor => cl::White,
    color => cl::Black,
    icon => Prima::Icon->load('icon.png'),
);

# You can load the standard info icon:
# my $logo = Prima::StdBitmap::icon(sbmp::Information);

# or load a custom icon
my $logo = Prima::Icon->load('tip.png');

# Insert the icon into the main window
$mw->insert( ImageViewer =>

    image => $logo,
    origin => [20, 200],
    size => [32, 32],
);

$mw->insert( Label =>

    text => "Did you know?",
    alignment => ta::Left,
    wordWrap => 1,
    origin => [70, 180],
    width => 250,
    height => 50,
    font => { size => 14, style => fs::Bold },
    color => cl::Black,
);


my %quotes = (

1 => "We don't stop playing because we grow old; " .
     "we grow old because we stop playing.\n\n" .
     "George Bernard Shaw",
2 => "Well done is better than well said.\n\n" .
     "Benjamin Franklin",
3 => "Tis better to have loved and lost " .
     "than never to have loved at all.\n\n" .
     "Alfred Lord Tennyson",
4 => "For every complex problem there is an answer " .
     "that is clear, simple, and wrong.\n\n" .
     "H. L. Mencken",
5 => "If the freedom of speech is taken away " .
     "then dumb and silent we may be led, " .
     "like sheep to the slaughter.\n\n" .
     "George Washington",
);

# Add a panel to act as a border around the label
my $panel = $mw->insert( 'Widget::Panel' =>

    origin => [20, 60],
```

```
    size => [300, 120],
    width => 300,
    height => 120,
    backColor => cl::LightGray,
    raise => 0,
    borderWidth => 1,
);

my $index = 1;

$panel = $mw->insert( Label =>

    origin => [25, 70],
    size => [285, 100],
    text => $quotes{$index},
    alignment => ta::Left,
    font => { size => 8, },
    wordWrap => 1,
    backColor => cl::LightGray,
    color => cl::Black,
);

$mw->insert( Button =>

    origin => [150, 10],
    size => [75, 30],
    text => "Next tip",
    onClick => sub {
                        ($index < (scalar keys %quotes)) ? $index++
                                                         : $index = 1;
                         $panel->text($quotes{$index});
                    },
);

$mw->insert( Button =>

    origin => [245, 10],
    size => [75, 30],
    text => "Exit",
    onClick => sub { exit; },
);

Prima->run;
```

Listing 10.3: Tip of the Day Application

10.4 Embedding images

In the previous code, we introduced the ImageViewer module, which displayed several quotes that could be navigated using the "Next Tip" button. In this chapter, we will delve deeper into the capabilities of the ImageViewer module.

10.4.1 A simple graphical window

Let’s first create a simple graphical window, displaying an image centered both horizontally and vertically. Instead of the origin property, the more efficient pack geometry manager is used to position the image. In Part 4, we will discuss this feature in detail. For now, this will suffice:

```
pack => { expand => 1, fill => 'both' }
```

Explanation:

- expand => 1: the widget will expand to fill any extra space in the parent container.
- fill => 'both': the widget will expand both horizontally and vertically to fill the available space.

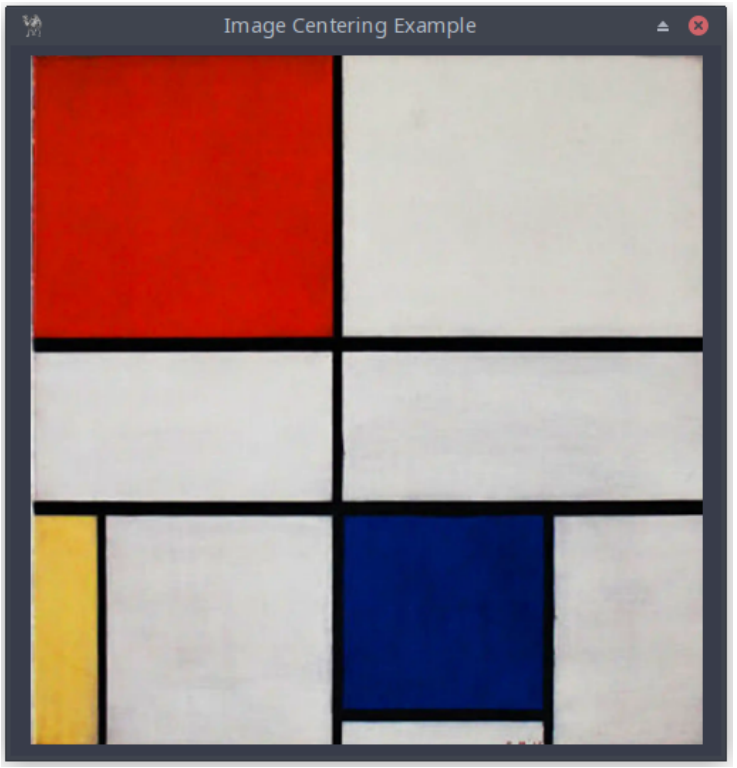


Figure 10.4: The Main Window, Presenting a Centered Image

```
use Prima qw(Application ImageViewer);

# load the image
my $image = Prima::Image->load('img/mondriaan.png') or
                          die "Cannot load image";

my $mw = Prima::MainWindow->new(

    text   => 'Image Centering Example',
    size  => [500, 500],
    centered => 1,
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
    icon  => Prima::Icon->load('icon.png'),
);

# create an ImageViewer widget to display the image

my $image_viewer = $mw->insert( ImageViewer =>

    pack => { expand => 1, fill => 'both' },

    # set the loaded image*
    image => $image,

    # center the loaded image vertically
    valignment => ta::Center,

    # center the loaded image horizontally
    alignment => ta::Center,
);

Prima->run;
```

Listing 10.4: The Main Window, Presenting a Centered Image

10.4.2 Adding and removing scrollbars

The easiest way to add scrollbars is by reducing the window size. Here, instead of [500, 500], we use [400, 400], and the scrollbars are automatically displayed.

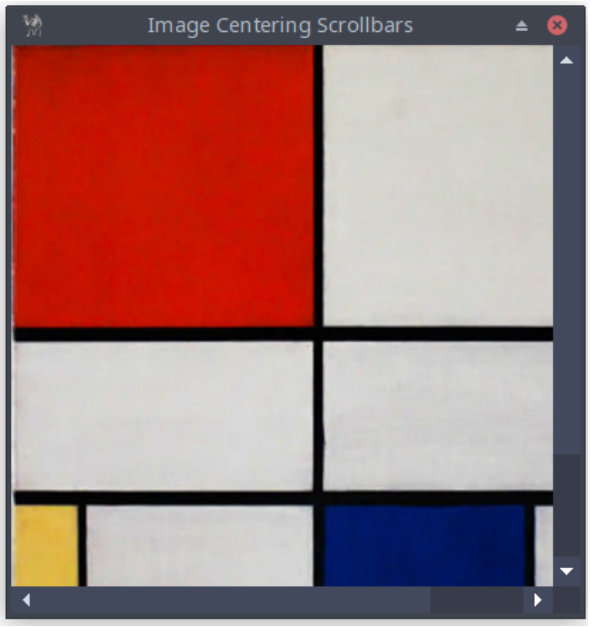


Figure 10.5: The Main Window, Presenting an Image with Scrollbars

To add scrollbars explicitly, use

```
vScroll => 1,
hScroll => 1,
```

To remove scrollbars explicitly, use

```
vScroll => 0,
hScroll => 0,
```

10.4.3 A slideshow

The following example is a simple slideshow that uses the ImageViewer, Label, and Button classes. New is the property pad in

```
pack => { expand => 0, fill => 'x', pad => 10, }
```

which simply means that a padding of 10 pixels is added around the widget. `fill => 'x'` means that the widget will expand horizontally to fill the available space in the parent container.

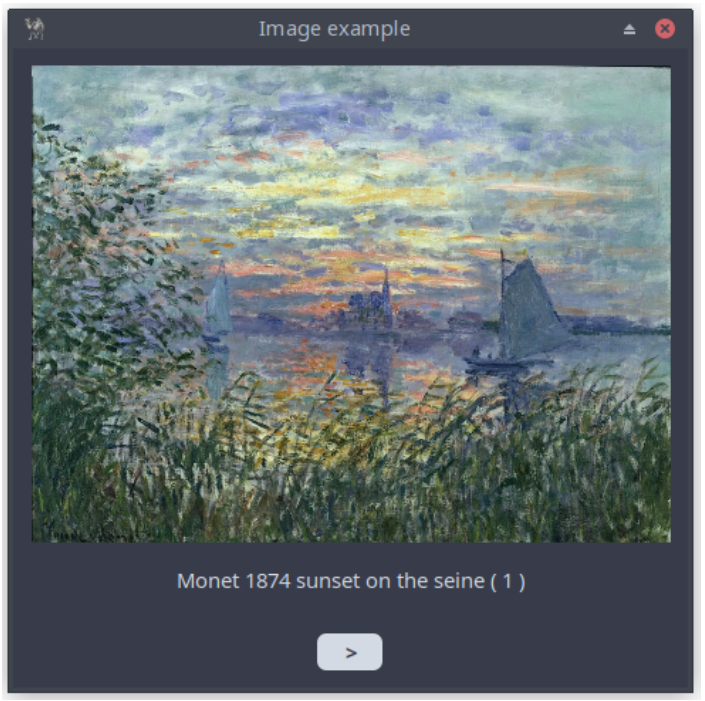


Figure 10.6: The Main Window, Presenting a Slideshow of Images

```
use Prima qw(Application ImageViewer Label Buttons);

my @images = ( 'img/monet_1874_sunset_on_the_seine.png',
               'img/monet_1882_sunset_at_pourville.png',
               'img/monet_1883_sunset_at_etretat.png',
               'img/monet_1886_sunset_at_giverny.png'
);

my @copy_images = @images;
my @texts;
my $no = 0;

# extract description form filenames

foreach (@copy_images) {

    $_ =~ s/img\// /g;
    $_ =~ s/_|\.png/ /g;
    $_ =~ s/^\s+//;
    $no++;
    $_ .= "( $no )";
    $_ = ucfirst($_);
    push(@texts, $_);
};

my $mw = Prima::MainWindow->new(
    text => 'Image example',
    size => [475, 450],
    centered => 1,
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
    icon => Prima::Icon->load('img/icon.png'),
);

my $current_index = 0;
my $image = Prima::Image->load($images[$current_index]) or
                          die "Cannot load image";

# Prima::ImageViewer - image, icon, and bitmap viewer

my $image_viewer = $mw->insert( ImageViewer =>

    pack => { expand => 1, fill => 'both', pad => 0 },
    image => $image,
    valignment => ta::Center,
    alignment => ta::Center,
);

my $label = $mw->insert( Label =>

    pack => { expand => 0, fill => 'x', pad => 10, },
    height => 30,
    font => { size => 11 },
    text => $texts[$current_index],
    alignment => ta::Center,
    valignment => ta::Top,
);


sub show_next {
```

```
    # the expression ($current_index + 1) % @images is used to increment
    # the current index and ensure it wraps around when it reaches the
    # end of the @images array

    $current_index = ($current_index + 1) % @images;
    $image->load($images[$current_index]) or die "Cannot load image";
    $image_viewer->image($image);
    $label->text($texts[$current_index]);
}


my $next_button = $mw->insert( Button =>

    pack => { expand => 0, fill => 'none', pady => 20 },
    width => 50,
    height => 30,
    text => ' > ',
    default => 1,
    onClick => sub { show_next(); },
);

Prima->run;
```

Listing 10.5: The Main Window, Presenting a Slideshow of Images

11. Layout Management Fundamentals

11.1 An Introduction to Geometry Managers

In Prima, geometry managers control how widgets are positioned within their containers. Unlike absolute positioning with the `origin` property, geometry managers provide flexible ways to organize widgets that automatically adapt to window resizing and content changes.

11.2 Absolute Positioning with `origin`

Before exploring geometry managers, let's review the basic positioning method we've used so far:

```
$mw->insert( Button =>
    origin => [50, 75],
    size => [150, 50],
    text => 'Click me',
);
```

The `origin` property specifies the widget's position relative to its parent container, with coordinates [x, y] where:

- x: horizontal position from the left edge
- y: vertical position from the bottom edge

From Chapter 10's random number example:

```
my $label = $mw->insert( Label =>
    origin => [110, 125],
    # ...
);

$mw->insert( Button =>
    origin => [50, 75],
    # ...
);
```

While simple, this approach has limitations:

1. Requires manual recalculation when window size changes
2. Doesn't automatically adjust to content changes
3. Makes complex layouts difficult to maintain

11.3 Using the *pack* Geometry Manager Effectively

The pack geometry manager provides a more flexible alternative:

```
$container->insert( Widget =>
    pack => {
        side => 'top',   # Position relative to container
        pad => 5,        # Padding around widget
        fill => 'x',     # Fill horizontally
        expand => 0      # Don't expand to fill space
    },
    # Other widget properties...
);
```

Key pack options:

1. **side**: Determines widget position
    - 'top' (default), 'bottom', 'left', 'right'
2. **fill**: Controls widget expansion
    - 'none' (default), 'x', 'y', or 'both'
3. **expand**: Allocates extra space
    - 0 (false) or 1 (true)
4. **pad**: Adds padding around widget
    - Single value or [horizontal, vertical]

Example application:

```
$mw->insert( Button =>
    text => 'Next tip',
    pack => {
        side => 'bottom',
        pad => 10,
        fill => 'x',
    },
);

$mw->insert( Label =>
    text => $quotes{$index},
    pack => {
        side => 'top',
        pad => 5,
        fill => 'both',
        expand => 1,
    },
);
```

This creates a layout where:

1. The label expands to fill available space
2. The button stays at the bottom
3. Automatic padding maintains spacing

The pack manager automatically handles window resizing and maintains the relative positions of widgets.

12. Managing User Input

12.1 InputLine Basics

The InputLine widget provides powerful single-line text input capabilities with extensive customization options. This section introduces the core properties and methods that control text input behavior.

Property	Description
autoHeight BOOLEAN	If 1, adjusts height when font changes (Default: 1)
readOnly BOOLEAN	If 1, text cannot be edited (Default: 0)
passwordChar CHARACTER	Character shown instead of text when writeOnly=1 (Default: *)
writeOnly BOOLEAN	If 1, hides input showing passwordChar (Default: 0)
insertMode	Manages typing mode (1=insert, 0=overwrite) (Default: Toggle with Insert key)

Table 12.1 Properties InputLine

Default methods with shortcuts:

- Copy (Ctrl+C)
- Cut (Ctrl+X)
- Delete (Del)
- Paste (Ctrl+V)
- Select all (Ctrl+A)

Here's a concise password application demonstrating InputLine properties:

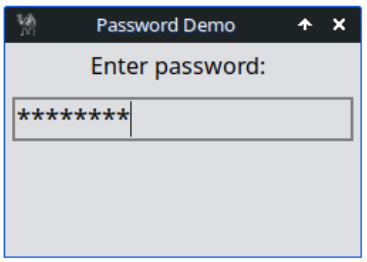


Figure 12.1: Password Field with Masked Characters

```
use Prima qw(Application InputLine Label);

my $mw = Prima::MainWindow->new(
    text => 'Password Demo',
    size => [250, 150],
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
);

# set icon if available; fastest option with minimal error handling
$mw->icon(Prima::Icon->load('icon.png')) if -r 'icon.png';

$mw->insert( Label =>
    pack => { side => 'top', pad => 10 },
    text => 'Enter password:',
    font => { size => 12 },
);

my $password = $mw->insert( InputLine =>
    pack => { fill => 'x', side => 'top', pad => 10 },
    text => '',
    writeOnly => 1,          # Hide actual input
    passwordChar => '*',     # Show asterisks instead
    borderWidth => 2,        # Thicker border
    font => { size => 14 }, # Larger text
    maxLen => 20,            # Limit length
    autoSelect => 0,         # Don't select all on focus
    readOnly => 0,           # Allow editing
);

# Set focus to password field
$password->focused(1);

Prima->run;
```

Listing 12.1: Password Field with Masked Characters

12.2 Validation Patterns

InputLine widgets support *onChange* events that let you validate user input in real-time. The following example demonstrates validating text input to ensure only alphabetic characters are entered (just for fun: enter a number). For the colored instructions, *Prima::Drawable::Markup* is used. Some text markup to the Label widget is added.

The application:

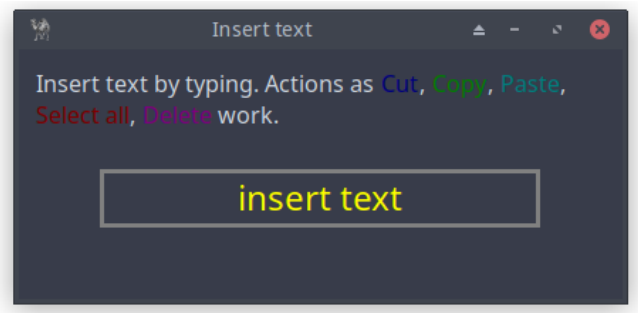


Figure 12.2: Validated Text Input with Visual Feedback

```
use Prima qw(Label Buttons InputLine Application);
use Prima::Drawable::Markup q(M);

my $mw = Prima::MainWindow->new(
    text => 'Insert alfabetic text',
    size => [425, 225],
    icon => Prima::Icon->load('icon.png'),
);

# C<Blue|Cut> is the markup code to color the text Cut blue
# see https://metacpan.org/pod/Prima::Const#cl%3A%3A-colors
my $label_text = "Insert text (letters only) by typing. Actions as " .
                 "C<Blue|Cut>, C<Green|Copy>, C<Cyan|Paste>, " .
                 "C<Red|Select all>, C<Magenta|Delete> work.";
$mw->insert( Label =>

    pack       => { fill => 'x', side => 'top', pad => 25,  },
    # backslashs \ is necessary for markup text; alternative: the character M
    text       => \ $label_text,
    wordWrap   => 1,
    font       => { size => 12, },
    autoHeight => 1,
);

my $input = $mw->insert( InputLine =>
    pack    => { fill => 'none', side => 'top', pad => 25 },
    text    => 'insert text',
    color   => 0xFFFF00,
    width   => 310,
    # ta::Left, ta::Right, ta::Center; default value: ta::Left
    alignment => ta::Center,
    font    => { size => 18 },
    # if 1, all the text is selected when the widget becomes focused.
    autoSelect => 0,
    # width of the border around the widget; default value: depends on the skin
    borderWidth => 3,
    # The maximal length of the text, that can be stored into text or typed
    # by the user. Default value: 256
    maxLen  => 50,
    # no right mouse-click: see section 12.3
    popupItems => [],

    # blink signals 'invalid' input
    # $_[0]->backColor(0xFF0000); sets background to red color
    # just for fun: enter a number...
    onChange => sub {

        my $text = $_[0]->text;
        if ($text =~ /[0-9]/) {
            $_[0]->backColor(0xFF0000);
            # blink when any numbers are detected
            $_[0]->blink;
        }
        else {
                $_[0]->backColor(cl::White);
        }
    },
);

# set the cursor at the end
$input->charOffset(length($input->text));
```

```
$input->focused(1);
Prima->run;
```

Listing 12.2: Validated Text Input with Visual Feedback

The *onChange* event handler checks for numeric characters and provides visual feedback when invalid input is detected. This pattern can be adapted for various validation requirements.

A more sophisticated validation example will be discussed in Part 8: Subclassing and Overriding methods.

12.3 Working with Selection and Cursor Control

The selection property allows you to manage selected text within an InputLine widget. This is particularly useful when building applications that require text manipulation.

The *selection* property, defined by START and END is introduced, specifies the beginning and the end of the selected text. If no text is selected, **START** and **END** will have the same value. The related properties **selStart** and **selEnd** also manage the selection range.

The following example demonstrates a self-assessment exercise: the user selects a word, mentally translates it into Latin, and then checks the answer. The score is tracked throughout.

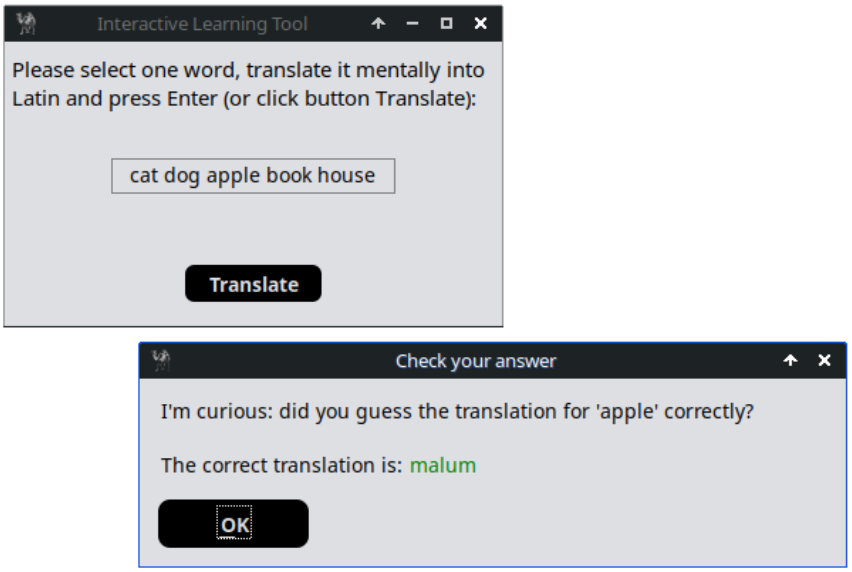


Figure 12.3: Selectable Text with Translation

```
use Prima qw(Application InputLine Buttons MsgBox);
use Prima::Drawable::Markup q(M);

# translation hash (from English to Latin for demonstration)
my %translation = (
    'cat'     => 'felis',
    'dog'     => 'canis',
    'apple'   => 'malum',
    'book'    => 'liber',
    'house'   => 'domus',
);

my $mw = Prima::MainWindow->new(
    text  => 'Interactive Learning Tool',
    size  => [350, 200],
    icon => Prima::Icon->load('icon.png'),
);

my $label_text = "Please select one word, translate it mentally into " .
                 "Latin and press Enter (or click button Translate):";

my $label = $mw->insert( Label =>
    pack   => {fill=>'both', side => 'top', padx => 10, pady => 20},
    text   => $label_text,
    wordWrap => 1,
    autoHeight => 1,
    font   => {size => 11},
);

my $input = $mw->insert(InputLine =>
    pack   => {side => 'top', pad => 40},
    size   => [200, 25],
    text   => 'cat dog apple book house',
    # if 1, all the text is selected when the widget becomes focused
    autoSelect => 0,
    # if 1, the text cannot be edited by the user
    readOnly => 1,
    alignment => ta::Center,
    popupItems => [],
);

my $translate_button = $mw->insert( Button =>
    pack   => {side => 'bottom', pad => 30 },
    text    => 'Translate',
    size    => [100, 30],
    default => 1,
    growMode => gm::Center,
    onClick => sub { translate_selected_text(); },
);

sub translate_selected_text
{
    my $message;
    my ($sel_start, $sel_end) = $input->selection;
    if ($sel_start != $sel_end)
    {
       my $selected_word = substr($input->text, $sel_start,
                                                $sel_end-$sel_start);
       # trim selection
       $selected_word =~ s/(^\s+|\s+$)//g;
```

```
        if (exists $translation{$selected_word}) {
          # Show the correct translation in a message box, the Latin word
          # in Green
          my $translation = $translation{$selected_word};
          my $text = "I'm curious: did you guess the translation for " .
                     "'$selected_word' correctly?\n\n" .
                     "The correct translation is: C<Green|$translation>";
          my $user_input = Prima::MsgBox::message_box(
                           'Check your answer', M $text, mb::Ok,
          );

        }
        else {
                  message_box( 'Warning!',
                               'Please select only one word before translating!',
                               mb::Ok, compact => 1,);
        }
    }
    else {
              message_box( 'Warning!',
                           'Please select one word before translating!',
                            mb::Ok, compact => 1,);
    }
}
Prima->run;
```

Listing 12.3: Selectable Text with Translation

12.4 Adding Context Menus

You can customize the right-click context menu using the *popupItems* property:

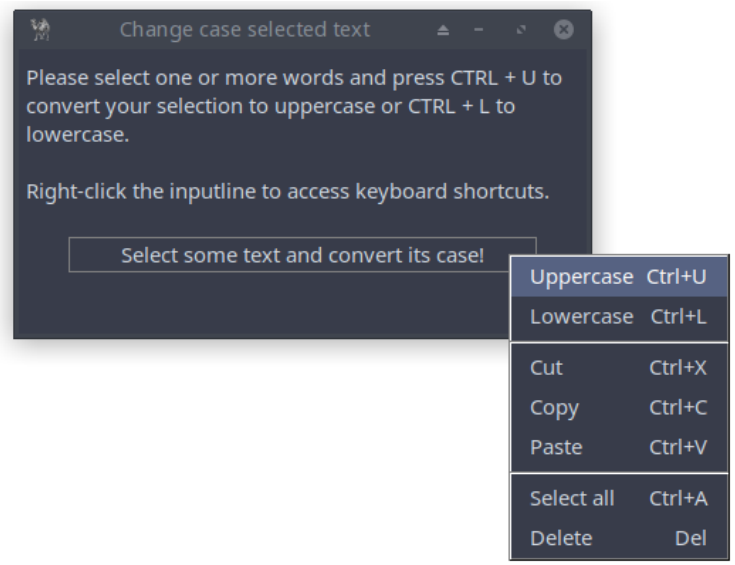


Figure 12.4: Custom Context Menu

```
use Prima qw(Application Label InputLine Buttons Menus MsgBox);

my $mw = Prima::MainWindow->new(
    text => 'Change case selected text',
    size => [400, 200],
    icon => Prima::Icon->load('icon.png'),
);

my $label = $mw->insert( Label =>
    pack   => {fill=>'both', side => 'top', padx => 10, pady => 20},
    text   => "Please select one or more words and press CTRL + U to " .
              "convert your selection to uppercase or CTRL + L to " .
              "lowercase.\n\nRight-click the inputline to access " .
              "keyboard shortcuts.",
    wordWrap => 1,
    autoHeight => 1,
    font   => {size => 11},
);

my ($input, $btn);

$input = $mw->insert( InputLine =>
    pack   => {fill => 'none', side => 'top', padx => 10, pady => 20},
    text    => 'Select some text and convert its case!',
    alignment => ta::Center,
    size    => [330, 25],
    popupItems  => [

      ['Uppercase' => 'Ctrl+U'    => '^U', sub{ change_case($input, "u") }],
      ['Lowercase' => 'Ctrl+L'    => '^L', sub{ change_case($input, "l") }],
      [],
      ['Cut'        => 'Ctrl+X'    => '^X', sub{ $_[0]->cut }],
      ['Copy'       => 'Ctrl+C'    => '^C', sub{ $_[0]->copy }],
      ['Paste'      => 'Ctrl+V'    => '^V', sub{ $_[0]->paste }],
      [],
      ['Select all' => 'Ctrl+A'    => '^A' => sub { $_[0]->select_all }],
      ['Delete'     => 'Del'       => kb::NoKey, sub{ $_[0]->delete }],
    ],

);

sub change_case {
    my ($input, $case) = @_;
    my ($sel_start, $sel_end) = $input->selection;

    # only manipulate if text is selected
    if ($sel_start != $sel_end) {
        # get the entire text
        my $current_text = $input->text;

        # get the selected text
        my $selected_text = substr($current_text, $sel_start,
                                   $sel_end-$sel_start);

        # replace the selected text with its uppercase version
        substr($current_text, $sel_start, $sel_end - $sel_start,
                                    uc($selected_text)) if ($case eq "u");
        substr($current_text, $sel_start, $sel_end - $sel_start,
                                    lc($selected_text)) if ($case eq "l");

        # update the InputLine with the new text
```

```
        $input->text($current_text);

    } else {

        Prima::MsgBox::message_box('Warning!', 'No text selected!',
                                                mb::Ok, compact => 1,);
    }
}
Prima->run;
```

Listing 12.4: Custom Context Menu

The *popupItems* property accepts an array reference containing menu item definitions. Each item consists of a display name and associated action.

We added two new ones (uppercase and lowercase) and defined the shortcuts and actions explicitly, for example:

['Copy' => 'Ctrl+C' => '^C', sub{ $_[0]->copy }],

which associates the ‘copy’ action with a subroutine that calls the ‘copy’ method on the widget (*$_[0]*), here the InputLine widget. It’s not necessary, because ‘copy’ is a built-in method. So this will work:

['Copy' => 'Ctrl+C' => '^C' => 'copy'],

To assign 'Copy' to another, unnecessary shortcut Ctrl+D (D could stand for Duplicate), you could write:

['Copy' => 'Ctrl+D' => '^D' => 'copy' }],

12.5 Undo/Redo Built- In Features

Implementation programmatic undo/redo support is easy:

```
popupItems  => [
                   ['Undo' => 'Ctrl+Z' => '^Z', sub{ $_[0]->undo }],
                   ['Redo' => 'Ctrl+Y' => '^Y', sub{ $_[0]->redo }],
               ],
```

12.6 Numeric Spin Boxes

The program shows a combination of a inputline and a pair of arrow buttons, allowing users to input a numerical value in an intuitive way. It provides functionality to either type the number directly or adjust it incrementally by clicking the up or down arrows. This control is useful in scenarios where a bounded numerical input is required, such as temperature conversion from Fahrenheit to Celsius.

This control called SpinCtrl is part of the *wxPython* library, a popular toolkit for creating graphical user interfaces (GUIs) in Python but not in Perl Prima.

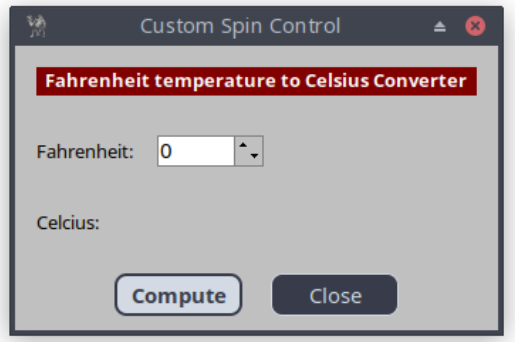


Figure 12.5: Number Field with Spin Buttons

```
use Prima qw(Application Label Buttons Sliders);

my $mw = Prima::MainWindow->new(
    text => 'Custom Spin Control',
    size => [340, 200],
    backColor => cl::LightGray,
    icon => Prima::Icon->load('icon.png'),
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
);

$mw->insert ( Label =>
    origin => [15, 165],
    size   => [310, 20],
    text   => 'Fahrenheit temperature to Celsius Converter',
    alignment => ta::Center,
    font   => { size => 10, style => fs::Bold },
    color  => cl::White,
    backColor => cl::Red,
);

$mw->insert ( Label =>
    origin => [15, 115],
    size   => [100, 20],
    text   => 'Fahrenheit:',
    alignment => ta::Left,
    font   => { size => 10, },
    color  => cl::Black,
);

my $inputline = $mw->insert( SpinEdit =>
    origin => [100, 115],
    width  => 75,
    min => -459,
    max => 1000,
    ownerBackColor => 0,
    editProfile => {color => cl::Black, backColor => cl::White},
    spinProfile => {backColor => cl::LightGray, color => cl::Black},
);

$mw->insert ( Label =>
    origin => [15, 40],
    size   => [60, 45],
    text   => 'Celcius:',
    alignment => ta::Left,
    font   => { size => 10, },
    color  => cl::Black,
),

my $result = $mw->insert( Label =>
    origin => [100, 40],
    size   => [60, 45],
    text   => '',
    alignment => ta::Left,
    color  => cl::Blue,
);

$mw->insert( Button =>
    origin => [70, 10],
    size => [90, 30],
    text => 'Compute',
    default => 1
```

```
    default => 1,
    onClick => sub { on_compute(); },
);

$mw->insert( Button =>
    origin => [180, 10],
    size => [90, 30],
    text => 'Close',
    onClick => sub { $::application->destroy() },
);

sub on_compute {
        my $fahr = $inputline->text;
    my $cels = (($fahr - 32) * 5 / 9);
    # round: 2 digits after decimal point
    my $factor = 10**2;
    $cels = int( $cels * $factor + 0.5 ) / $factor;
    $result->text($cels);
}

Prima->run;
```

Listing 12.5: Number Field with Spin Buttons

12.7 Built- In Color Selector

This Prima-based color selector allows users to adjust Red, Green, and Blue (RGB) values through sliders, with real-time feedback on color changes. It displays the selected color in a preview area and updates both the RGB and hexadecimal values for easy reference.

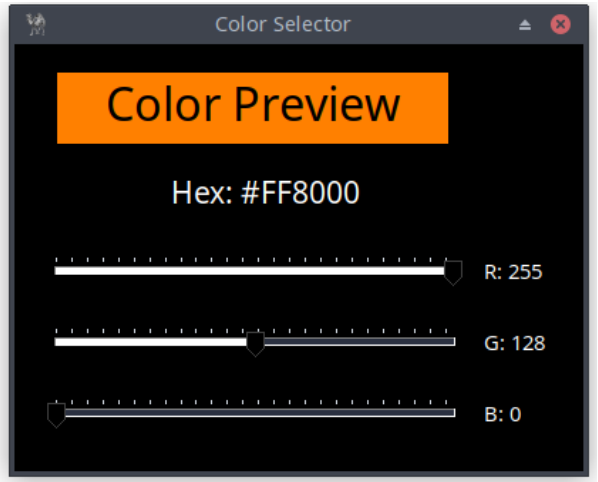


Figure 12.6: RGB Color Selection Interface

```
use Prima qw(Application Label Sliders);

my $mw = Prima::MainWindow->new(
    text        => 'Color Selector',
    size        => [400, 300],
    backColor   => cl::Black,
    color       => cl::Black,
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu | bi::TitleBar,
);

# Try loading icon but don't die if missing
my $icon = eval { Prima::Icon->load('icon.png') };
$mw->icon($icon) if $icon;

my %slider_defaults = (
    size => [300, 30],
    # if 1, the widget is drawn vertically, and the slider moves from bottom
    # to top. If 0, the widget is drawn horizontally, and the slider moves
    # from left to right.
    vertical => 0,
    # sets the lower limit for value; default: 0
    min => 0,
    # sets the upper limit for value; default: 100
    max => 255,
    # selects an integer value between min and max and the corresponding
    # sliding bar position
    value => 128,
    # if 1, the parts of the shaft are painted with different colors to
    # increase visual feedback. If 0, the shaft is painted with the single
    # default background color
    ribbonStrip => 1,
    onChange    => sub { update_color() },
);

my $color_display = $mw->insert( Label =>
    text       => 'Color Preview',
    origin     => [30, 230],
    size       => [275, 50],
    alignment  => ta::Center,
    backColor  => cl::White,
    font       => { size => 24 },
);

my $hex_display = $mw->insert( Label =>
    text      => 'Hex: #808080',
    origin    => [110, 180],
    size      => [360, 30],
    alignment => ta::Left,
    font      => { size => 16 },
    color     => cl::White,
);

my %label_defaults = ( size => [50, 30], color => cl::White );

# Create RGB sliders and labels
my @slider_labels = ( ['R', 120], ['G', 70], ['B', 20] );
my @sliders;
my @labels;
foreach my $color_data (@slider_labels) {
    my ($label, $y) = @$color_data;
```

```
    push @sliders, $mw->insert( Slider =>
        origin => [20, $y],
        %slider_defaults,
    );
    push @labels, $mw->insert( Label =>
        text   => "$label: 128",
        origin => [330, $y],
        %label_defaults,
    );
}

# convert RGB values to packed 24-bit color integer
# format: 0xRRGGBB where RR=red, GG=green, BB=blue
sub RGB {
    # shift red 16 bits left, green 8 bits left, keep blue in lowest 8 bits
    ($_[0] << 16) | ($_[1] << 8) | $_[2]
}

sub update_color {
    my @vals = map { $_->value } @sliders;   # (r, g, b) values 0-255
    my ($r, $g, $b) = @vals;

    # set preview background using packed RGB value
    $color_display->backColor( RGB($r, $g, $b) );

    # format RGB values as 2-digit hex for display
    my $hex = sprintf("#%02X%02X%02X", $r, $g, $b);
    $hex_display->text("Hex: $hex");

    # update each label with its corresponding color value
    for my $i (0 .. $#labels) {
        $labels[$i]->text("$slider_labels[$i][0]: $vals[$i]");
    }
}

update_color();

Prima->run;
```

Listing 12.6: RGB Color Selection Interface

13. Selection Widgets

13.1 Radio Buttons

Allows users to select one option from a set of mutually exclusive choices. Appears in sets of two or more, with each option represented by a small circle and label; a filled circle indicates the selected option. Ideal for forms, settings, or tests requiring a single selection.

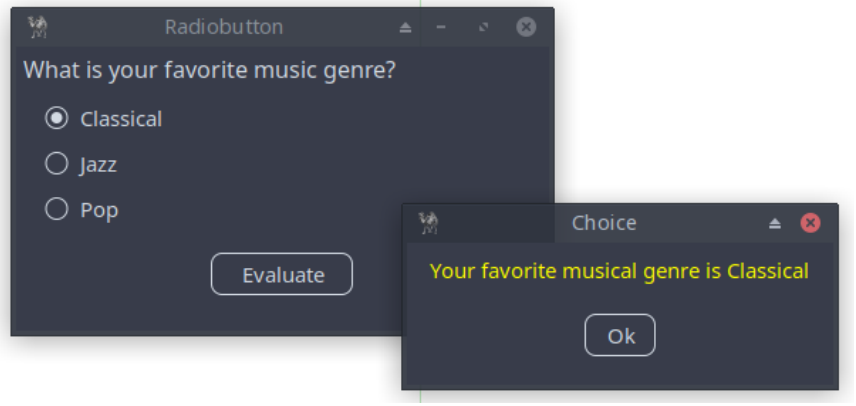


Figure 13.1: Application with Radio Buttons

```
use Prima qw(Application Buttons Label MsgBox);
use Prima::Drawable::Markup q(M);

my $mw = Prima::MainWindow->new(
    text=> 'Radiobutton',
    size => [375, 200],
    icon => Prima::Icon->load('icon.png'),
);

$mw->insert( Label =>
    pack => {fill => 'x', side => 'top', pad => 10},
    text  => 'What is your favorite music genre?',
    font  => {size => 12,},
);

my $choice;

my $grp = $mw->insert( GroupBox =>
    text => '',
    origin => [20,70],
    onRadioClick => sub { $choice = $_[1]->text; }
);

# fill => 'x', expand => 1 to align the radiobuttons
$grp->insert( Radio =>
    pack => {side => 'top', fill => 'x', expand => 1}, text => $_)
            for (qw(Classical Jazz Pop));

# check the first option*
$grp->index(0);

$mw->insert( Button =>
    pack => { fill => 'none', side => 'bottom', pad => 50 },
    size => [100, 30],
    text => 'Evaluate',
    onClick => sub {
                message_box(
                    'Choice',
                    M "C<Yellow|Your favorite musical genre is $choice>",
                    mb::Ok, compact => 1,
                );
    }
);

Prima->run;
```

Listing 13.1: Application with Radio Buttons

Tips:

1. Replace `origin => [20,70]` by `pack => {side => 'top',}` to center the radiobuttons.
2. To add a tooltip to radio buttons, use the `hint` property. This property lets you define a text message that appears when users hover over the widget. Tooltips are useful for providing additional context or guidance, for example:

```
$grp->insert( Radio =>
    pack => { side => 'top', fill => 'x', expand => 1},
              text => 'Classical', hint => 'Bach, Brahms, Bruckner');
$grp->insert( Radio =>
    pack =>
       {side => 'top', fill => 'x', expand => 1}, text => $_) for qw(Jazz Pop);
```

13.2 Checkboxes

A Checkbox allows users to make a binary choice, such as yes/no or true/false. It includes a small box with a label, where a check mark appears when the option is chosen. If the box is empty, the option is unselected. Checkboxes are commonly used in forms or settings to enable options or toggle features on and off.

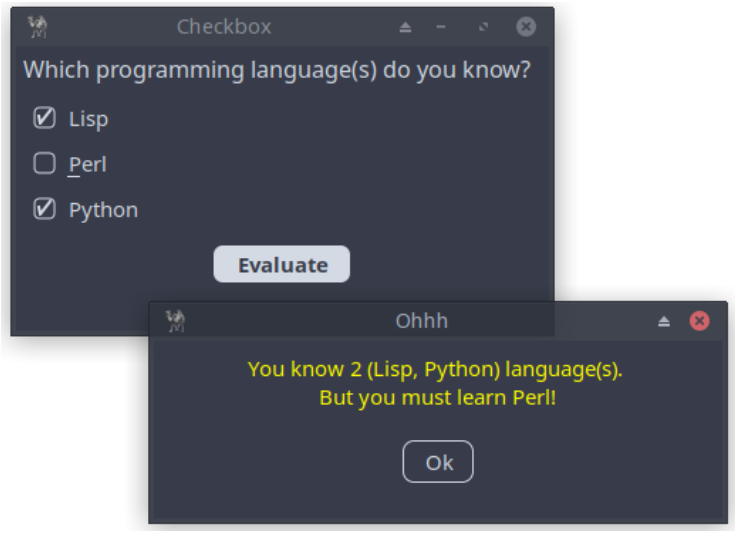


Figure 13.2: Application with Checkboxes

```
use Prima qw(Application Buttons Label MsgBox);

my $mw = Prima::MainWindow->new(
    text=> 'Checkbox',
    size => [375, 200],
    icon => Prima::Icon->load('icon.png'),
);

my $counter = 0;

sub EventHandler {
     my ($var) = @_;
     ($var->checked) ? $counter++ : $counter--;
}

$mw->insert( Label =>
          pack => { fill => 'x', side => 'top', pad => 10 },
          text   => 'Which programming language(s) do you know?',
          autoHeight => 1,
          font   => { size => 12, },
);

my($chk_lisp, $chk_perl, $chk_python);

$chk_lisp = $mw->insert( CheckBox =>
    pack => { fill => 'x', side => 'top', padx => 25 },
    text => 'Lisp',
    onClick => sub { EventHandler($chk_lisp); },
    # hint displays a tooltip when the mouse pointer is hovered over the
    # widget longer than some timeout, until the pointer is drawn away.
    hint => "I like Lisp, \nespecially the dialect newLisp!",
);

$chk_perl = $mw->insert( CheckBox =>
    pack => { fill => 'x', side => 'top', padx => 25 },
    # the character after ~ (tilde) character, is treated as a hot key,
    # and the character is underlined. If the user presses the corresponding
    # character key then Click event is called
    text => '~Perl',
    onClick => sub { EventHandler($chk_perl); },
    # right mouse click!
    popupItems => [ ['Perl is a powerful, feature-rich programming language
                      developed over 30 years.' => ''] ],
);

$chk_python = $mw->insert( CheckBox =>
    pack => { fill => 'x', side => 'top', padx => 25 },
    text => "Python",
    onClick => sub { EventHandler($chk_python); },
);

$mw->insert( Button =>
    pack => { fill => 'none', side => 'top', pad => 15 },
    size => [100, 30],
    text => 'Evaluate',
    backColor => 0xcccccc,
    color => 0x000000,
    # user can press the enter key
    default => 1,
    growMode => gm::Center,
    onClick => sub {
```

```
        my @checked_options;

        # check the state of each checkbox
        push (@checked_options, $chk_lisp->text) if $chk_lisp->checked;
        push (@checked_options, substr($chk_perl->text, 1))
                                                 if $chk_perl->checked;
        push (@checked_options, $chk_python->text)
                                                 if $chk_python->checked;

        # print the result based on the checked options
        if (@checked_options) {
            my $str = $counter . " (" .  join(", ", @checked_options) . ")";

            message_box('Ohhh', "You know $str language(s). \n
                                 But you must learn Perl!",
                                 mb::Ok, compact => 1,)
                                     if (! $chk_perl->checked);

            message_box('Congratulations', "You know $str language(s). \n\n
                                            Indeed, Perl is a great language!",
                                            mb::Ok, compact => 1,)
                                                if ($chk_perl->checked);
        }
        else {
                 message_box('Ahhh', "No options selected",
                                     mb::Ok, compact => 1,);
        }
    }

);

Prima->run;
```

Listing 13.2 Application with Checkboxes

13.3 Combo Boxes

If you are giving the user a small list of choices then stick with the radiobuttons. However, use the combobox for long lists. A user can select one option. I prefer the type of combobox with an instruction inside as a header.

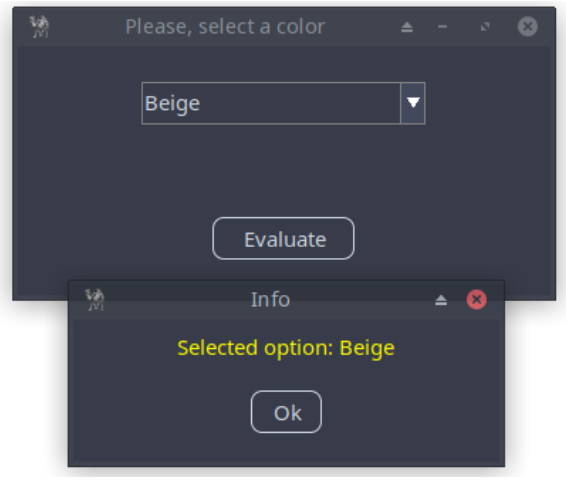


Figure 13.3: Application with Combobox

```
use Prima qw(Application Label Buttons ComboBox MsgBox);

my $mw = Prima::MainWindow->new(
      text=> 'Combobox',
      size => [375, 175],
      icon => Prima::Icon->load('icon.png'),
      skin => 'flat',
);

# If the collection in a combo box is long enough, a scroll bar will
# appear to accommodate it.

my $combo = $mw->insert( ComboBox =>
     pack => { fill => 'none', side => 'top', pad => 50 },
     size => [200, 30],
     name => 'Please, select a color',
     items => [ 'Apricot', 'Beige', 'Black', 'Blue', 'Brown', 'Cyan',
                'Green', 'Grey', 'Lavender', 'Lime', 'Magenta' ],
     # select first item
     text => 'Apricot',
     # the combo box should function as a drop-down list: you can type
     # in your own text, in addition to selecting items from the drop-down.
     style => (cs::DropDownList),
);

$mw->insert( Button =>
    pack => {fill => 'none', side => 'bottom', pad => 50},
    size => [100, 30],
    text => 'Evaluate',
    onClick => sub {
                         message_box("Info",
                                     "Selected option: " . $combo->text,
                                     mb::OK, compact => 1);
                   }
);

Prima->run;
```

Listing 13.3: Application with Combobox

13.4 List Boxes

If you are giving the user a small list of choices then stick with the checkbox. However, use the listbox for long lists. A user can select one or more options.

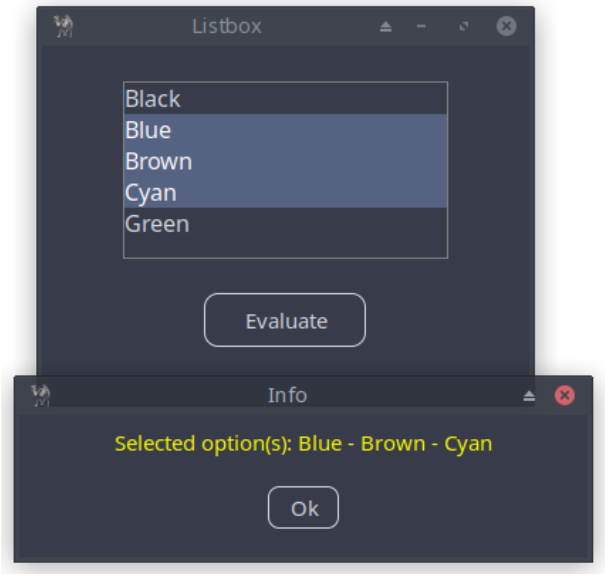


Figure 13.4: Application with Listbox

```
use Prima qw(Application Buttons Label MsgBox ComboBox);

my $mw = Prima::MainWindow->new(
      origin  => [100, 100],
      size    => [300, 200],
      # designScale: width and height of the font when designed,
      # transformed in proportion between the designed and the
      # actual font metrics, to avoid garbled windows on different systems
      designScale => [7, 16],
      text => "Listbox",
      icon => Prima::Icon->load('icon.png'),
);

my $list = $mw->insert( ListBox =>
      name           => 'ListBox1',
      origin         => [50, 80],
      size           => [200, 100],
      # if 0, the user can select only one item
      multiSelect    => 1,
      # if 1, the user can drag mouse or use Shift key plus arrow keys to
      # perform range selection;
      # the Control key can be used to select individual items.
      extendedSelect => 1,
      font           => {size => 12},
      items          => [ 'Black', 'Blue', 'Brown', 'Cyan', 'Green' ],
      align          => ta::Left,

);

$mw->insert( Button =>
     origin => [100, 30],
     size => [100, 30],
     text => 'Evaluate',
     onClick => sub {

         my $SelectedOptions = "";

         foreach (@{ $list->selectedItems }){

             if ($SelectedOptions eq "") {
                 $SelectedOptions = $list->items->[$_];
                 # alternative: $SelectedOptions = $list->{items}[$_];
                 # alternative: $SelectedOptions = $list->{items}->[$_];
             }
             else {
                 $SelectedOptions = join (" - ",
                     $SelectedOptions, $list->items->[$_]);
                     # alternative:
                     # $SelectedOptions = join (" - ", $SelectedOptions,
                     #                          $list->{items}[$_]);
                     # alternative:
                     # $SelectedOptions = join (" - ", $SelectedOptions,
                     #                          $list->{items}->[$_]);
             }
         };
         $SelectedOptions = "no selection!" if (! $SelectedOptions);
         message_box('Info', "Selected option(s): " . $SelectedOptions,
                                                      mb::Ok, compact => 1,);
     }
);
```

```
Prima->run;
```

Listing 13.4: Application with Listbox

13.5 CheckLists

In this example, you'll learn how to create a simple checklist application using the `CheckList` widget from the `ExtLists` module. For a list with this many options, checkboxes are not ideal. A `CheckList` keeps the interface tidy and clear. Users can see all the options in one compact view, toggle selections on/off without confusion, and no complex multi-selection behavior is needed like holding Ctrl or Shift in a ListBox.

This is ideal for applications that offer a wide range of customizable settings.

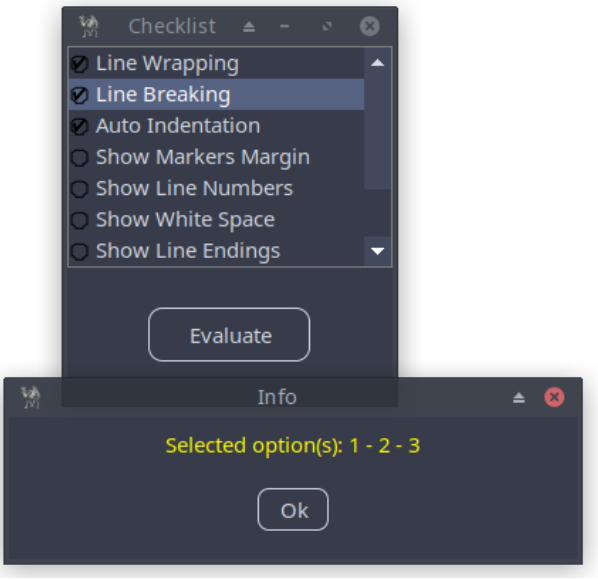


Figure 13.5: Application Featuring Checklist

```
use Prima qw(ExtLists Application Buttons Label MsgBox);

my $mw = Prima::MainWindow->new(
    size => [ 200, 200],
     # Ensures consistent widget scaling across different screen
     # resolutions and DPIs
    designScale => [7,16],
    text => "Checklist",
    icon    => Prima::Icon->load('icon.png'),
);

my @items = ('Line Wrapping', 'Line Breaking', 'Auto Indentation',
             'Show Markers Margin', 'Show Line Numbers', 'Show White Space',
             'Show Line Endings', 'Show Message Window', 'Show Tool Bar',
             'Show Side Bar');

# Initialize the bit vector with items 1, 3, 5, and 7 checked
my $v = '';
# Item 1 checked
vec($v, 0, 1) = 1;
# Item 3 checked
vec($v, 2, 1) = 1;
# Item 5 checked
vec($v, 4, 1) = 1;
# Item 7 checked
vec($v, 6, 1) = 1;

# or in one statement: vec($vec, 0, 8) = 0x55; The hexadecimal
# value 55 is 01010101 in binary.
# this means that alternating bits (1, 3, 5, 7) are set to 1,
# while bits (2, 4, 6, 8) are set to 0.

# alternatively, using binary directly: $v = pack("B*", '01010101');

my $checklist= $mw->insert( CheckList =>
    pack     => { fill => 'both', expand => 1},
    items    => \@items,
     # initial bit vector
    vector   => $v,
);

$mw->insert( Button =>
     pack     => { fill => 'none', side => 'top', expand => 1},
     size => [100, 30],
     text => 'Evaluate',
      # event handler for when an item is clicked
     onClick => sub {

         my $selected = '';

         for(my $i = 0; $i < scalar(@items); $i++) {
             # get the updated vector after the click event: $checklist->vector
             # get the bit value: vec($checklist->vector, $i, 1) where 1
             # if checked, 0 if unchecked
             $selected .= ($selected ? ' - ' : '') . ($i + 1) if ( vec($checklist->vector, $i, 1) );
         }

         $selected = "no selection!" if (! $selected);
         message_box('Info', "Selected option(s): " . $selected,
                             mb::Ok, compact => 1,);            }
);
```

```
Prima->run;
```

Listing 13.5: Application Featuring Checklist

`ExtLists` is a subclass of `Lists`. That means that some inherited properties from the `Lists` class, like the `multiColumn` property, are particularly useful. When set to 1 (`multiColumn => 1`), you'll see the following:

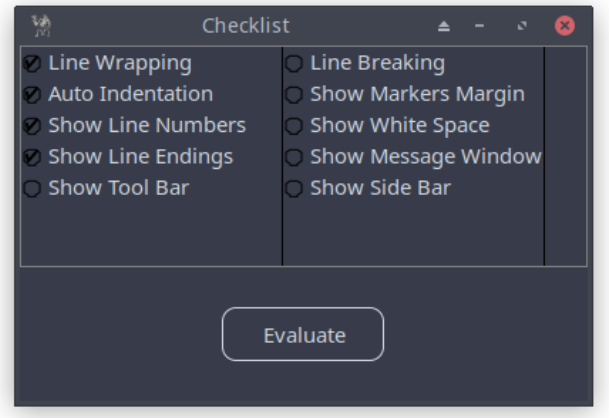


Figure 13.6: Application Featuring Checklist and multicolumn => 1

Add `vertical => 1` and you'll get:

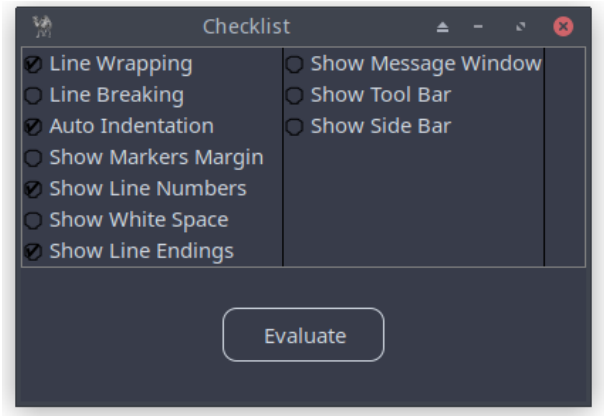


Figure 13.7: Application Featuring Checklist and multicolumn => 1 and vertical => 1

13.6 Draggable Items in Lists

KnowledgeSorter is an educational app that challenges users to arrange items in the correct order, making it perfect for subjects like history, science, programming, and more. This can be easily implemented in Prima by using the `draggable` property in the ListBox widget.

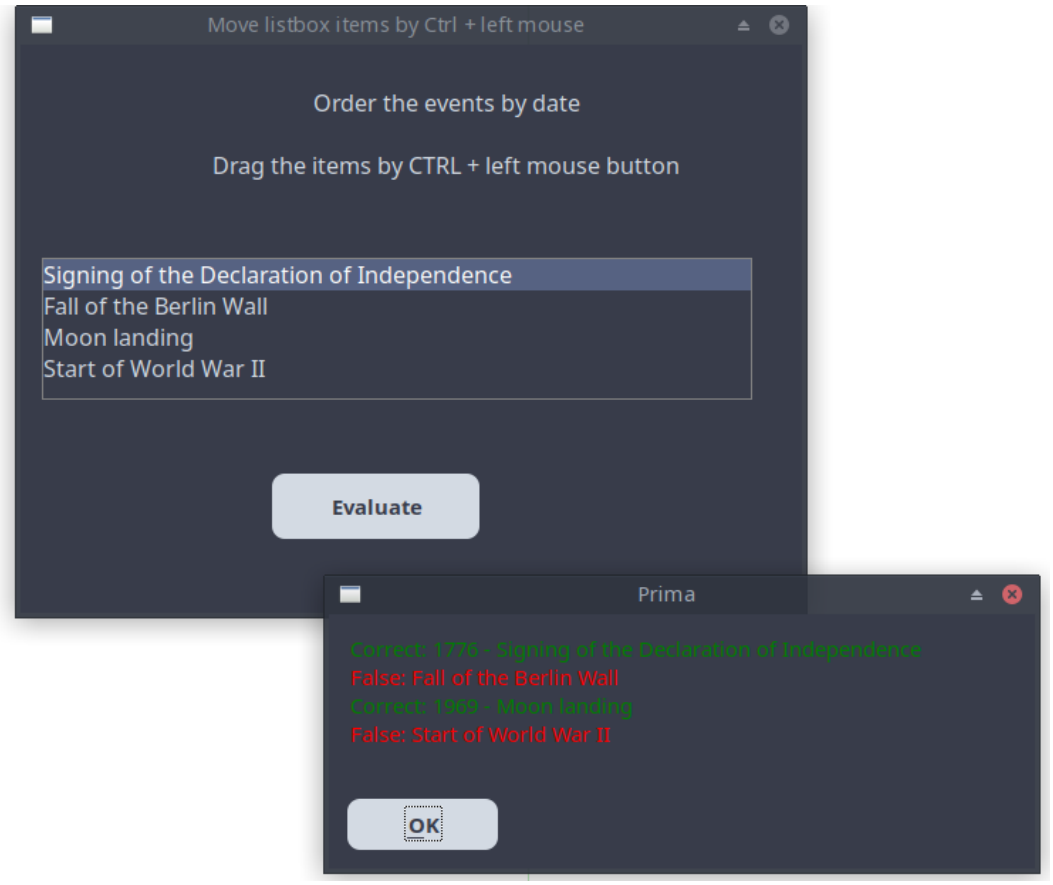


Figure 13.8: Application Using the Draggable Property in the ListBox Widget

```
use Prima qw(Application Buttons Label ComboBox MsgBox);
use Prima::Drawable::Markup q(M);

my $mw = Prima::MainWindow->new(
    text        => 'Move listbox items by Ctrl + left mouse',
    size        => [530, 400],
    font        => { size => 12, },
    borderStyle => bs::Dialog,
    borderIcons => bi::All & ~bi::Maximize,

);

my $in = $mw->insert( Label =>
    origin      => [15, 300],
    size        => [500, 70],
    text        => "Order the events by date\n\n
                    Drag the items by CTRL + left mouse button",
    alignment   => ta::Center,
    autoHeight  => 1,

);

my $lb = $mw->insert( ListBox =>
    origin => [15, 150],
    size   => [500, 100],
    font   => { size => 12},
    items  => [ 'Fall of the Berlin Wall',
                'Signing of the Declaration of Independence',
                'Moon landing',
                'Start of World War II' ],
    draggable  => 1,
    align      => ta::Left,
);


$mw->insert( Button =>
    origin  => [175, 50],
    size    => [150, 50],
    text    => 'Evaluate',
    default => 1,
    onClick => sub { check_order($lb); },
);

sub check_order
{

    my @arr = @{ $lb->items};

    my $str = '';

     # colored text by C<color|text>
     # read https://metacpan.org/pod/Prima::Drawable::Markup

    ($arr[0] eq "Signing of the Declaration of Independence")
               ? $str .= "C<Green|Correct: 1776 - $arr[0]\n"
               : $str .= "C<LightRed|False: $arr[0]\n";

    ($arr[1] eq "Start of World War II")
               ? $str .= "C<Green|Correct: 1939 - $arr[1]\n"
               : $str .= "C<LightRed|False: $arr[1]\n";

    ($arr[2] eq "Moon landing")
```

```
    ($arr[2] eq "Moon Landing")
              ? $str .= "C<Green|Correct: 1969 - $arr[2]\n"
              :  $str .= "C<LightRed|False: $arr[2]\n";

    ($arr[3] eq "Fall of the Berlin Wall")
              ? $str .= "C<Green|Correct: 1989 - $arr[3]\n"
              : $str .= "C<LightRed|False: $arr[3]\n";

    $str .= ">\n";

    # https://metacpan.org/pod/Prima::MsgBox
    # message displays 'Prima' as the window title
    message( M "$str", mb::Ok);
}

Prima->run;
```

Listing 13.6: Application Using the Draggable Property in the ListBox Widget

Study the alternative code of sorting-quiz.pl, which you can find in the `example` directory of the Prima distribution.

Closing Words

In this part, you learned the essential widgets that form the heart of any Prima application - buttons, labels, text inputs, and selection tools. You also discovered how to lay out these elements cleanly and how to make them interactive and user-friendly.

With these fundamentals, you can now build interfaces that respond to user actions, display information clearly, and guide input in reliable ways. These widgets are the core vocabulary of every larger application you will create.

Part 4 - Dialogs and User Interaction

14. Standard and Custom Dialogs

Part 4 introduces Prima's dialog system: message boxes, file dialogs, the Find/Replace dialog, and other built-in tools. You also learn the difference between modal and modeless dialogs, and how to build a simple custom dialog for your own input tasks.

14.1 Message Boxes (Info, Warning, Confirm)

Use *`Prima::MsgBox`* for simple alerts, confirmation boxes, and questions:

```
use Prima qw(Application MsgBox);

message_box( "Information", "Operation completed.",
             mb::Ok, compact => 1);

my $answer = message_box(
                 "Exit?", "Close application?",
                 mb::OKCancel | mb::Question, compact => 1 );

exit if( $answer == mb::OK );

Prima->run;
```

Listing 14.1: Messagebox

Common flags:

- Buttons: `mb::Ok`, `mb::OKCancel`, `mb::YesNo`
- Icons: `mb::Information`, `mb::Error`, `mb::Warning`, `mb::Question`
- Option: `compact => 1` reduces the dialog width when possible, removing unused space.

14.2 Open/Save Dialogs

Prima provides flexible open/save dialogs:

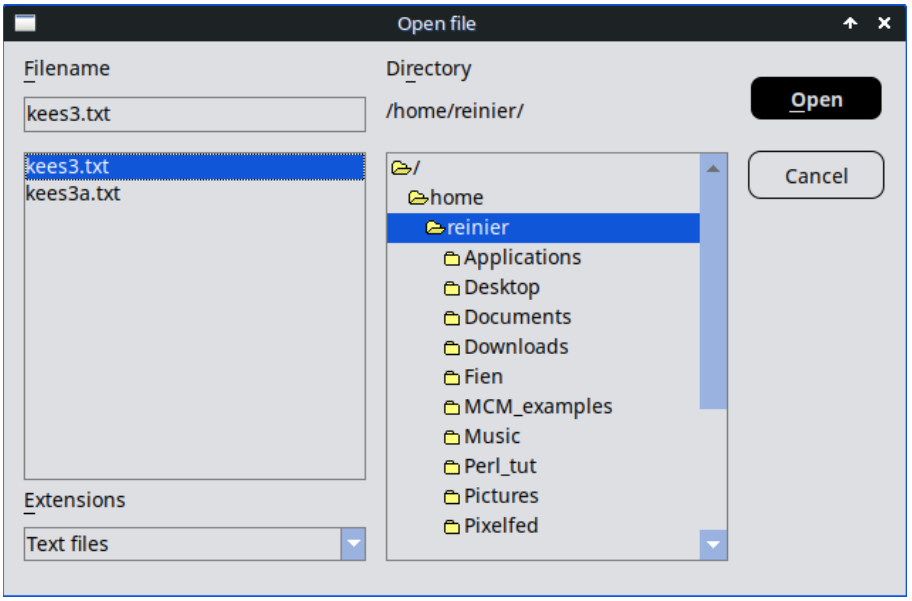


Figure 14.1: File Open dialog

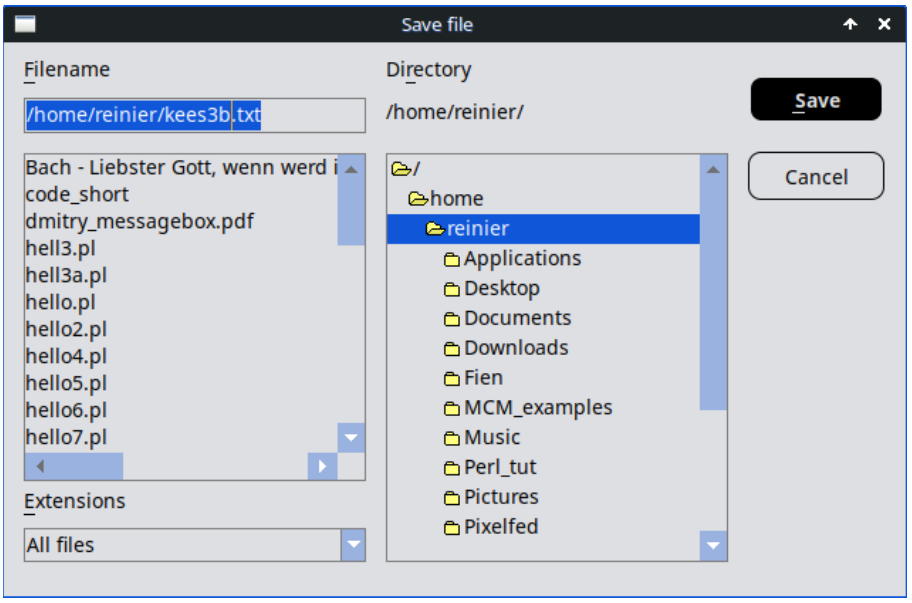


Figure 14.2: File Save dialog

```
use Prima qw(Application Dialog::FileDialog);

#-------------------------------------------------------------
# Open file dialog
#-------------------------------------------------------------

my $open = Prima::Dialog::OpenDialog->new(
    filter => [
        [ "Text files" => "*.txt" ],
        [ "All files"  => "*"      ],
    ],
);

unless ( $open->execute ) {
    message_box(
        "Exit", "No file selected!",
        mb::OK | mb::Warning,
        compact => 1
    );
    $::application->close;
}

open( my $fh, '<', $open->fileName ) or do {
    message_box(
        "Error", "Cannot open '".$open->fileName."': $!",
        mb::OK | mb::Error,
        compact => 1
    );
    $::application->close;
};

# local $/ = undef; means slurp mode, i.e. reading an entire
# file into a single string all at once, here: $content
my $content = do { local $/; <$fh> };
close $fh;

#-------------------------------------------------------------
# Save file dialog
#-------------------------------------------------------------

my $save = Prima::Dialog::SaveDialog->new(
    fileName => $open->fileName,
);

unless ( $save->execute ) {
    message_box(
        "Exit", "Save cancelled!",
        mb::OK | mb::Warning,
        compact => 1
    );
    $::application->close;
}

open( my $out, '>', $save->fileName ) or do {
    message_box(
        "Error", "Cannot write to '".$save->fileName."': $!",
        mb::OK | mb::Error,
        compact => 1
    );
    $::application->close;
};
```

```
close $out;

message_box(
    "Done",
    "File saved successfully.",
    mb::OK,
    compact => 1
);

$::application->close;

Prima->run;
```

Listing 14.2: File Open dialog and File Save Dialog

Key options:

- *directory* - initial directory
- *filter* - filter sets
- *text* - window title

14.3 Find and Replace Dialogs

Perl Prima includes a built-in Search and Replace dialog. While we'll explore this in Part 9, you don't have to wait to see it in action! Simply run *editor.pl* from the examples directory of the Prima package to try it out for yourself.

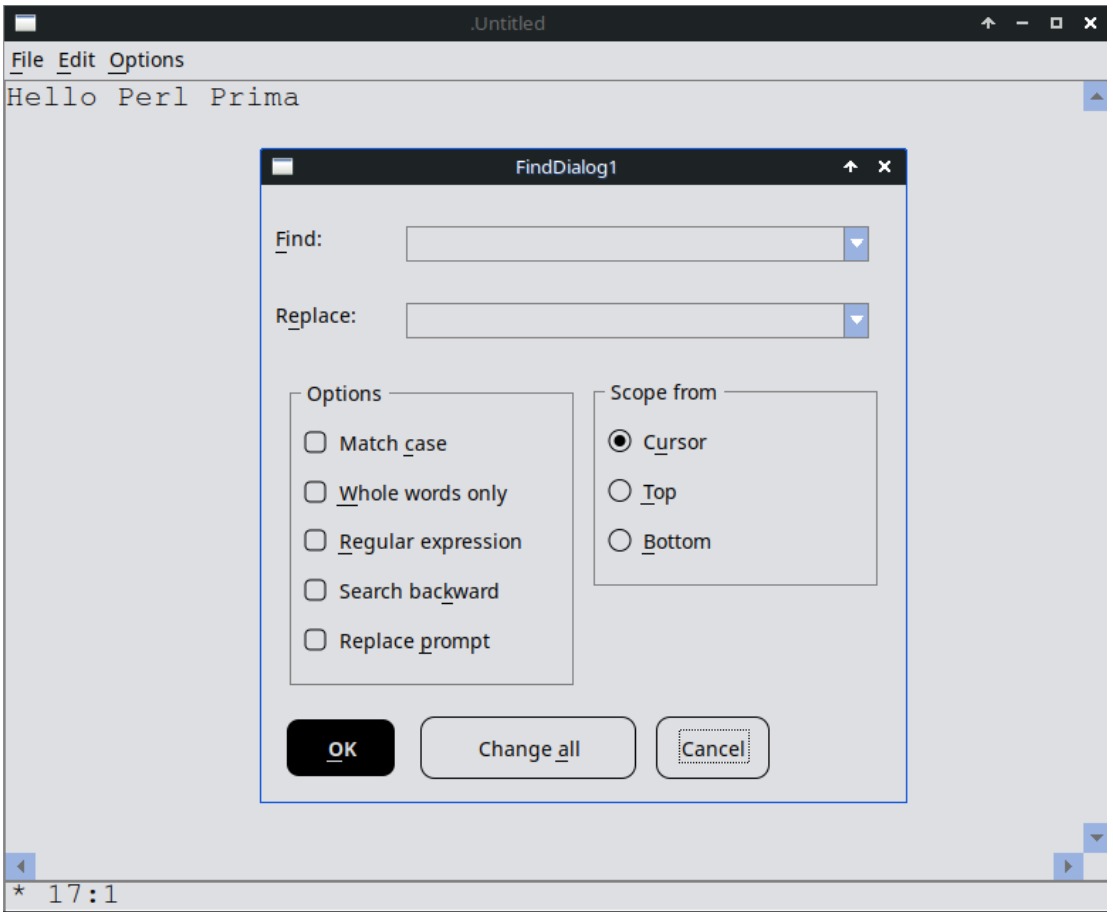


Figure 14.3: Find and Replace dialog

14.4 More Dialogs in Prima

Prima provides several built-in dialogs that simplify common user interactions. These dialogs offer a familiar, consistent interface across applications and can be integrated with only a few lines of code.

Module	Description	Example
FontDialog	Standard font selection dialog	Prima::Dialog::FontDialog
ImageDialog	Image file open dialog	Prima::Dialog::ImageDialog
ImageSaveDialog	Image save dialog	Prima::Dialog::ImageDialog
ColorDialog	Standard color selection	Prima::Dialog::ColorDialog
PrintDialog	Standard printer setup dialog	Prima::Dialog::PrintDialog

Table 14.1: Dialogs in Prima

14.5 Modal vs. Modeless Dialogs

A modal dialog blocks the rest of the application until the dialog is closed. Nothing else can be clicked or interacted with.

```
my $dlg = Prima::Dialog->new(
    text => "Modal Example",
);
$dlg->execute; # 'execute' opens the dialog modally (blocks until closed).
```

A modeless dialog does *not* block:

```
my $dlg = Prima::Dialog->new(
    text => "Modeless Window",
);
$dlg->show; # show displays the dialog modelessly
            # (the application continues running).
```

Use modal dialogs for confirmations, warnings, and file open/save operations. Use modeless dialogs for tools, inspectors, and utility windows.

14.6 Constructing Your Own Dialog

A custom dialog in Prima is simply a small modal window built from standard widgets and tailored to a specific task. Unlike general-purpose dialogs such as file pickers or message boxes, a custom dialog lets you design your own layout and collect exactly the information you need. By subclassing `Prima::Dialog`, inserting controls such as labels, input fields, and buttons, and then running it with `execute`, you create a self-contained interaction that returns a result to the calling code. The following example shows a minimal custom dialog with an Input field, a Label and OK/Cancel buttons.

This example shows how to build a simple custom dialog and how to use it in the main program.

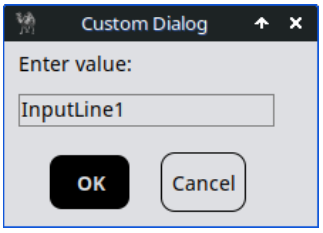


Figure 14.4: Custom Dialog

a. Beginner-First Version: Simple Custom Dialog

Start with the basics: a tiny dialog that just asks for a name and shows it back. This introduces the idea without overwhelming details like inheritance or key handling. Once you get this working, you can copy-paste it into your main program and see a window pop up.

```
use Prima qw(Application Buttons InputLine Label);

# A custom dialog
my $dlg = Prima::Dialog ->new(
    text => 'Custom Dialog',
    size => [200, 130],
    icon => Prima::Icon->load('icon.png'),
);

# Label: tells the user what to do
$dlg->insert( Label =>
    text   => "Enter value:",
    origin => [10, 100],
    size   => [280, 20],
);

$dlg->{input}  = $dlg->insert( InputLine =>
    origin => [10, 70],
    width  => 180,
);

# OK Button
# modalResult tells the dialog 'I was closed by OK'.
$dlg->insert( Button =>
    text        => "OK",
    default     => 1,
    origin      => [30, 10],
    width       => 60,
    modalResult => mb::OK,
);

# Cancel Button
# modalResult signals 'Cancel was pressed'.
$dlg->insert( Button =>
    text        => "Cancel",
    origin      => [110, 10],
    width       => 60,
    modalResult => mb::Cancel,
);

# Show the dialog and wait until the user clicks something.
my $res = $dlg->execute;
# After the dialog closes, check HOW it was closed:
if( $res == mb::OK ) {
    # OK -> read the text field we stored in $dlg->{input}->text
    print "You entered: " . $dlg->{input}->text ."\n";
} else {
    print "Cancelled.\n";
}

$::application->close;

Prima->run;
```

Listing 14.3: Custom Dialog

No fancy subclassing yet; we're just using Prima's built-in `Dialog` class like a blank canvas.

b. Advanced-Later Version: Full Custom Dialog with Subclassing

Now that the simple version works, upgrade it to a reusable "class" (like a blueprint you can create multiple times). This adds inheritance (`use base`), a setup method (`init`), and extras like ESC key support. It's more API-like but builds directly on the beginner code. I will be careful with you and annotate the code thoroughly. It won't be too difficult!

The code has two parts:

1. The Dialog (`MyDialog`)

Here we design the dialog: a label, an input field, and OK/Cancel buttons. For now, just see `$self` and `init()` as the standard way Prima lets us build our own dialog window. Later we'll look more closely at subclassing and how to make your own classes.

2. The Main Program

This part creates the dialog, shows it, and checks whether the user pressed OK or Cancel. If OK was clicked, we read the text from the input field.

```
use Prima qw(Application Buttons InputLine Label Const MsgBox);

# ------------------------------------------------------------
# Custom Dialog Class (Don't worry — you can treat it like a box
# we can 'design' once and then create later.)
# ------------------------------------------------------------
{
    # This tells Perl: 'MyDialog behaves like a Prima Dialog.'
    # (You don’t have to understand inheritance yet.)
    package MyDialog;
    use base qw(Prima::Dialog); # inherit from Prima::Dialog

    sub init{
        # Perl gives our method the object in $self or put simply:
        # $self is the dialog you are building.
        # Everything else (like text => 'Custom Dialog') goes
        # into %profile. We don’t have to worry about the details
        # yet — this is just Prima’s standard pattern.
        my ( $self, %profile ) = @_;
        # Always call the parent's init - standard Prima practice.
        $self->SUPER::init(%profile);

        # Label
        # A simple text line inside the dialog.
        $self->insert( Label =>
            text    => "Enter value:",
            origin  => [10, 100],
            width   => 150,
        );
        # A text field where the user can type.
        #
        # We also store a reference in $self->{input} so that
        # the main program can later read what the user typed,
        # publicly accessible as $dlg->{input}
        $self->{input} = $self->insert( InputLine =>
            origin => [10, 70],
            width  => 180,
        );

        # OK Button
        # modalResult tells the dialog 'I was closed by OK'.
        $self->insert( Button =>
            text        => "OK",
            default     => 1,
            origin      => [30, 10],
            width       => 60,
            modalResult => mb::OK,
        );

        # Cancel Button
        # Same idea: this signals 'Cancel was pressed'.
        $self->insert( Button =>
            text        => "Cancel",
            origin      => [110, 10],
            width       => 60,
            modalResult => mb::Cancel,
        );

        # ESC key closes dialog
        # This is just a convenience: press ESC → same as cancel.
        $self->onKeyDown( sub{
```

```
            my ($dlg, $code) = @_;
            # kb::Esc is provided by Prima::Const (imported via
            # Prima qw(... Const ...))
            $dlg->cancel if $code == kb::Esc;
        });

        return $self;
    }
}

# ------------------------------------------------------------
# Main Application
# ------------------------------------------------------------
# Create the dialog.
# (Prima will automatically call our init() above.)

my $dlg = MyDialog->new(
    text   => "Custom Dialog",
    width  => 220,
    height => 130,
    icon => Prima::Icon->load('icon.png'),
);

# Show the dialog and wait until the user clicks something.
my $res = $dlg->execute;
# After the dialog closes, check HOW it was closed:
if( $res == mb::OK ) {
    # OK -> read the text field we stored in $dlg->{input}
    my $value = $dlg->{input}->text;
    print "You entered: $value\n";
} else {
    print "Cancelled.\n";
}

$::application->close;

Prima->run;
```

Listing 14.4: Custom Dialog

Closing Words

In Part 4, you've explored Prima's powerful dialog system, from standard message boxes and file dialogs to custom input forms. These tools are not just theoretical - they are essential building blocks for real-world applications, including the editor you'll create in later chapters.

- Standard Dialogs: You can now work with message boxes, File/Save dialogs, FontDialog, ImageDialog, ImageSaveDialog, ColorDialog, and PrintDialog. *(The Find/Replace dialog will be covered later.)*
- Modal vs. Modeless: You know when to block user interaction (modal) for focused tasks and when to allow multitasking (modeless), ensuring smooth and intuitive workflows in your applications.
- Custom Dialogs: You've learned to build your own dialogs - from simple input forms to reusable, subclassed components -giving you the flexibility to create specialized interfaces like settings panels or about dialogs for your application.

Looking Ahead

In Part 9, you'll dive deeper into these concepts and develop a fully functional editor that integrates everything you've learned. Until then, take the time to experiment with the dialogs and try creating your own custom dialog. The more familiar you become with these features, the more confident and prepared you'll be when building your complete application.

Up Next: My Personal Message Utility!

Extra: a personal utility

1. Why `wordWrap` Is the Best Solution for a Message Box That Fits the Text

When showing messages of different lengths, the goal is simple: the dialog should always fit the text nicely - not too wide, not too cramped. It might seem easy to size the box by counting how many characters the text has, but that doesn't work well because fonts aren't uniform. Some letters, like W, take up more space than i, and different fonts or sizes change how much room the same text needs.

The `wordWrap` feature solves this automatically. It measures the real width of each word in the current font and wraps lines so the text fits perfectly inside the dialog. Combined with `autoHeight => 1`, the label adjusts its height too, so longer messages simply make the box taller instead of overflowing.

This approach uses Prima's own layout system — the same one it uses to draw text — so the result is always accurate, clean, and consistent. In short, `wordWrap` makes your message box smart enough to shape itself around any text, without guessing or manual adjustments.

The built-in message_box with its compact property already does a great job for most cases, but it doesn't cover everything I need. That's why I created my own message box subroutine - one that's highly customizable… and yes, with `wordWrap`!

2. Creating, invoking and modifying My Custom Message Window

The showMessageDialog subroutine, which accepts five arguments, is my solution. I use it for displaying messages (info, warning, error) and as a monitoring tool during debugging.

First step

Define a subroutine `showMessageDialog` and save it in an separate file (e.g., showMessageDialog.pl). Remember to add 1; at the end.

```
sub showMessageDialog {

    # this subroutine is called with 5 arguments, which are passed to a list.
    my ($width, $height, $title, $message, $textalign) = @_;

    # ensuring default values
    $width  = $width  && $width > 0 ? $width  : 300;
    $height = $height && $height > 0 ? $height : 140;
    $title  ||= 'Message';
    $message ||= '';
    $textalign ||= ta::Center;    # other options: ta::Left, ta::Right

    my $icon = eval { Prima::Icon->load('icon.png') };
    # warn "Failed to load icon: $@" if $@;

    my $dlg = Prima::Dialog->new(
        size     => [$width, $height],
        text     => $title,
        centered => 1,
        icon     => $icon,
    );

    $dlg->insert( Label =>
        pack       => { fill => 'x', side => 'top', pad => 12 },
        alignment  => $textalign,
        autoHeight => 1,
        # wordWrap   => 1: no manual textWidth calculation is needed
        wordWrap   => 1,
        text       => $message,
        color      => cl::Blue,
    );

    $dlg->insert( Button =>
            pack    => { fill => 'none', side => 'bottom', pad => 15 },
            size    => [50, 30],
            text    => 'Ok',
            default => 1,
            onClick => sub { $dlg->destroy; },
    );

    # make Escape close the dialog
    $dlg->onKeyDown(sub {
        my ($self, $code, $key, $mod) = @_;
        if ($key == kb::Escape) {
            $dlg->destroy;
        }
    });

    # execute brings the widget in a modal state
    $dlg->execute();
}

# return a true value to the interpreter
1;
```

Second step

Invoke showMessageDialog.pl in your program.

```
use Prima qw(Application Buttons Label);
use lib '.';
require 'showMessageDialog.pl';

my $icon = eval { Prima::Icon->load('icon.png') };
# warn "Failed to load icon: $@" if $@;

my $mw = Prima::MainWindow->new(
    text => 'Demo messagewindow',
    size => [360, 220],
    centered => 1,
    icon => $icon,
);

$mw->insert( Button =>
    pack => { side => 'top', pad => 150 },
    size => [120, 40],
    text => 'Message',
    onClick => sub {

        my $text = "This is a demo text";
        # you can easily change the width (here: 320) and height
        # (here: 140) if the text is large.
        showMessageDialog( 320, 140, "Demo", $text, ta::Left );
    },
);

Prima->run;
```

Third step

- You can easily adjust the width and height for larger text.
- Use `\n` to insert a line break.
- Add a color parameter for text or background color
- Add `use Prima::Drawable::Markup q(M);` for colored text. Example:

```
  $text = "This is a C<Red|demo> text ";
  showMessageDialog( 320, 140, "Demo", M $text, ta::Left );
```

Part 5 - Dynamic and Timed Widgets

15. Time-Based and Status Widgets

In Part 5, we move beyond static user-interface elements and explore widgets that *respond* to time, events, and user interaction. Many applications rely on components that update themselves - countdowns, clocks, progress indicators -or controls that reflect and modify application state. This part introduces you to Prima's tools for implementing these dynamic behaviors. You'll learn how to use timers to refresh widgets, display the progress of background operations, and implement interactive toggles that control features, settings, and application modes. By the end of this part, you will be able to design rich, responsive interfaces that feel alive and intuitive.

15.1 Using the Time Widget and Timers

The *Widget::Time* control and Prima's *Timer* class make it easy to build time-based interfaces. In this example, we develop a **stopwatch** with Start, Stop, and Reset buttons, updated once per second.

We separate the implementation into clear chunks so the logic remains easy to follow.

Figure 15.1: Stopwatch with Widget::Time and Timer Class

15.1.1 Main Window Setup

We begin by creating the main window. The size is defined once so all widgets can reference it.

```
my $app_width  = 500;
my $app_height = 200;

my $mw = Prima::MainWindow->new(
    size        => [$app_width, $app_height],
    text        => 'Stopwatch',
    icon        => Prima::Icon->load('icon.png'),
    backColor   => cl::LightGray,
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu | bi::TitleBar,
);
```

15.1.2 The Time Display (Widget::Time)

The *Widget::Time* control shows **hh:mm:ss**. Note that Prima uses the time order **[seconds, minutes, hours]**, which is a little different from what you might expect.

```
my $size_widget = 180;

$mw->insert( 'Widget::Time' =>
    origin    => [($app_width/2) - ($size_widget/2), 130],
    size      => [$size_widget, 39],
    name      => 'Time',
    time      => [0,0,0], # [sec, min, hour]
    format    => 'hh:mm:ss',
    alignment => ta::Center,
    font      => { size => 26 },
    readOnly  => 1,
    enabled   => 0,
    disabledBackColor => cl::LightGray,
    disabledColor     => cl::Black,
);
```

This widget never receives keyboard input; the stopwatch updates it automatically.

15.1.3 Stopwatch State Machine

We keep the entire stopwatch state in one hash `%S` (from Stopwatch), making the logic clean and centralized.

```
my %S = (
    start_time   => undef,  # when counting started
    paused_from  => undef,  # when pause started
    paused_total => 0.0,    # accumulated pause time
    running      => 0,
);
```

The reset helper restores everything:

```
sub reset_stopwatch {
    %S = (
        start_time => undef,
        paused_from => undef,
        paused_total => 0.0,
        running => 0 );

    $mw->Time->time([0,0,0]);
    $mw->mcTimer->stop;

    $mw->start_button->enabled(1);
    $mw->start_button->focused(1);
    $mw->stop_button->enabled(0);
    $mw->reset_button->enabled(0);
}
```

15.1.4 Calculating and Updating the Display

Every tick, the elapsed time is computed based on whether the stopwatch is running or paused.

```
sub update_display {
    return unless $S{start_time};

    my $now = time();
    my $elapsed =
        $S{running} ? $now - $S{start_time} - $S{paused_total}
                    : ($S{paused_from}
                          ? $S{paused_from} - $S{start_time} - $S{paused_total}
                          : 0);

    $elapsed = 0 if $elapsed < 0;

    my $hour = int($elapsed / 3600);
    my $min  = int(($elapsed % 3600) / 60);
    my $sec  = int($elapsed % 60);

    $mw->Time->time([$sec, $min, $hour]);
}
```

15.1.5 The Timer Object

A Prima `Timer` calls `update_display()` once per second.

```
$mw->insert(Timer =>
    name    => 'mcTimer',
    timeout => 1000,
    onTick  => sub { update_display() },
);
```

15.1.6 Start, Stop, Reset Buttons

These handlers form the user interface:

```
my %actions = (
    exit_button => sub { $::application->close },

    start_button => sub {
        unless ($S{start_time}) {
            $S{start_time}   = time();
            $S{paused_total} = 0.0;
        }
        if (defined $S{paused_from}) {
            $S{paused_total} += time() - $S{paused_from};
            $S{paused_from} = undef;
        }

        $S{running} = 1;
        update_display();
        $mw->mcTimer->start;

        $mw->start_button->enabled(0);
        $mw->stop_button->enabled(1);
        $mw->stop_button->focused(1);
        $mw->reset_button->enabled(0);
    },

    stop_button => sub {
        return unless $S{running};

        $S{paused_from} = time();
        $S{running} = 0;
        $mw->mcTimer->stop;

        $mw->start_button->enabled(1);
        $mw->start_button->focused(1);
        $mw->reset_button->enabled(1);
    },

    reset_button => sub { reset_stopwatch() },
);
```

The buttons themselves are inserted into a container widget.

15.1.7 Full Program Stopwatch

```
use Prima qw(Buttons Application Widget::Time);
# Time::HiRes::time gives us sub-second precision timestamps as a
# single floating value
use Time::HiRes qw(time);

# size of application; defined once so we can reuse it consistently
my $app_width  = 500;
my $app_height = 200;

my $mw = Prima::MainWindow->new(
    size        => [$app_width, $app_height],
    text        => 'Stopwatch',
    icon        => Prima::Icon->load('icon.png'),
    backColor   => cl::LightGray,
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu | bi::TitleBar,
);

# size of the time display widget;
my $size_widget = 180;

# insert a Widget::Time control into the main window. Widget::Time displays
# an hour:minute:second value. IMPORTANT: it expects the time array as
# [sec, min, hour] (this order is how Prima defines the property).
$mw->insert('Widget::Time' =>
    origin         => [($app_width / 2) - ($size_widget / 2), 130],
    size           => [$size_widget, 39],
    name           => 'Time',
    time           => [0, 0, 0], # initial value [sec, min, hour]
    format         => 'hh:mm:ss', # formatting pattern for display
    borderWidth    => 0,
    alignment      => ta::Center,
    font           => { size => 26 },
    cursorVisible  => 0,
    readOnly       => 1,
    enabled        => 0, # disabled so user can't edit it
    # disabledBackColor COLOR
    # The color to be used instead of the value of the ::backColor property
    # when the widget is in the disabled state.
    disabledBackColor => cl::LightGray,
    # disabledColor COLOR
    # The color to be used instead of the value of the ::color property
    # when the widget is in the disabled state
    disabledColor     => cl::Black,
);

# --- Stopwatch state ---
# we keep all state in a single lexical hash %S so it's easy to reason about.
# keys:
# - start_time: epoch seconds (floating) when the stopwatch first started
# - paused_from: epoch seconds when a pause (stop) began
# - paused_total: total seconds spent paused (accumulated)
# - running: boolean flag (1 when stopwatch is running)
my %S = (
    start_time   => undef,
    paused_from  => undef,
    paused_total => 0.0,
    running      => 0,
);

# reset_stopwatch: helper to reset the state and UI back to initial values.
```

```
# a 'reset' should both change the timer value shown and stop any
# running timer loop (so the program doesn't keep updating the display).
sub reset_stopwatch {
    %S = ( start_time => undef, paused_from => undef, paused_total => 0.0, running => 0 );
    # Widget::Time expects [sec, min, hour]
    $mw->Time->time([0,0,0]);
    # stop the periodic Timer so it stops calling update_display
    $mw->mcTimer->stop;
    # enable/disable buttons to reflect the reset state
    $mw->start_button->enabled(1);
    # move keyboard focus to Start so a user can press Enter to begin
    $mw->start_button->focused(1);
    $mw->stop_button->enabled(0);
    $mw->reset_button->enabled(0);
}

# update_display: compute the elapsed time and write it to the Widget::Time display.
# Explanation of the math:
# - if running: elapsed = now - start_time - paused_total
# - if paused:  elapsed = paused_from - start_time - paused_total
# the paused_total variable already contains previous pause durations. When we
# resume, we add the current pause length to paused_total so the display never
# counts paused time.
sub update_display {
    # if never started, there is nothing to display
    return unless $S{start_time};
    my $now = time();

    # choose how to compute elapsed depending on running/paused state
    my $elapsed = $S{running}
       ? $now - $S{start_time} - $S{paused_total}
       : ($S{paused_from} ? $S{paused_from} - $S{start_time} - $S{paused_total}
                          : 0);

    # guard against tiny negative values due to timing precision
    $elapsed = 0 if $elapsed < 0;

    # break down into hours / minutes / seconds
    my $hour = int($elapsed / 3600);
    my $min  = int(($elapsed % 3600) / 60);
    my $sec  = int($elapsed % 60);

    # Widget::Time wants [sec, min, hour]
    $mw->Time->time([$sec, $min, $hour]);
}

# Timer: runs every 1000 ms (1 second) and updates the display for an
# hh:mm:ss display, one-second resolution is enough and saves CPU.
$mw->insert(Timer =>
    name    => 'mcTimer',
    timeout => 1000, # milliseconds
    onTick  => sub { update_display() },
);

# Buttons: definitions and layout
# we create a small button container widget and then insert four buttons
# into it
my @buttons = (
    { name => 'start_button', text => 'Start', enabled => 1 },
    { name => 'stop_button',  text => 'Stop',  enabled => 0 },
    { name => 'reset_button', text => 'Reset', enabled => 0 },
    { name => 'exit_button',  text => 'Exit',  enabled => 1 },
```

```
    { name => 'exit_button', text => 'Exit', enabled => 1 },
);

# a plain Widget acts as a simple container to pack buttons horizontally.
my $button_widget = $mw->insert(Widget =>
    origin    => [0, 10],
    size      => [$app_width, 100],
    backColor => cl::LightGray,
);

# Button action handlers
# each coderef manipulates the %S state and the UI to provide the expected
# behavior
my %actions = (
    exit_button => sub { $::application->close },

    # Start: either start fresh or resume from a paused state
    start_button => sub {
        # if we've never started (or have reset), record the start time
        unless ($S{start_time}) {
            $S{start_time} = time();
            # ensure previously paused time doesn't leak through
            $S{paused_total} = 0.0;
        }
        # if resuming after a pause, compute how long we were paused and
        # add to paused_total
        if (defined $S{paused_from}) {
            $S{paused_total} += time() - $S{paused_from};
            $S{paused_from} = undef; # clear the paused marker
        }
        $S{running} = 1;   # mark as running
        update_display();  # refresh immediately so user sees instant response
        $mw->mcTimer->start; # start the periodic updates

        # update button states: Start disabled while running; Stop enabled
        $mw->start_button->enabled(0);
        $mw->stop_button->enabled(1);
        # move keyboard focus to Stop so pressing Enter will stop the stopwatch
        $mw->stop_button->focused(1);
        $mw->reset_button->enabled(0);
    },

    # Stop: pause the stopwatch without losing elapsed time
    stop_button => sub {
        # if already paused or never started, do nothing
        return unless $S{running};
        # mark the time the pause began
        $S{paused_from} = time();
        $S{running} = 0;
        $mw->mcTimer->stop; # stop periodic updates to save CPU

        # update buttons: start available to resume; reset becomes available
        $mw->start_button->enabled(1);
        # move keyboard focus back to Start so Enter resumes
        $mw->start_button->focused(1);
        $mw->reset_button->enabled(1);
    },

    reset_button => sub {
        reset_stopwatch();
    },
);
```

```
# insert button widgets into the container using simple packing rules
foreach my $btn (@buttons) {
    $button_widget->insert(Button =>
        pack    => { fill => 'x', side => 'left', pad => 15 },
        name    => $btn->{name},
        text    => $btn->{text},
        enabled => $btn->{enabled},
        onClick => $actions{$btn->{name}},
    );
}

Prima->run;
```

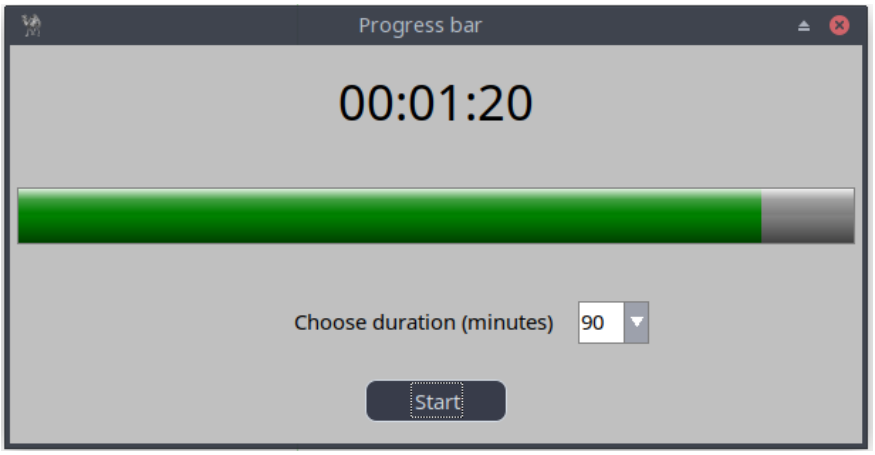
Listing 15.1: Stopwatch with Widget::Time and Timer Class

15.2 Progress Bars for Background Tasks

This example builds a **countdown timer** with:

- A dropdown to choose duration
- A progress bar that changes color (green → orange → red)
- A digital time display
- A Start/Stop toggle button
- A popup dialog at completion

The code is long, so here it is in digestible chunks.



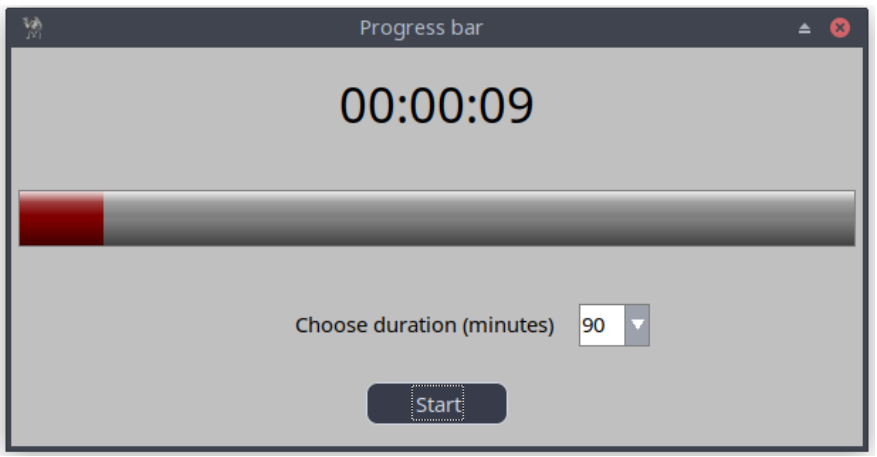


Figure 15.2: Timer Class and Progressbar

15.2.1 Window and Time Display

```
my $app_width = 600;

my $mw = Prima::MainWindow->new(
    text      => 'Progress bar',
    size      => [$app_width, 280],
    backColor => cl::LightGray,
    icon      => Prima::Icon->load('icon.png'),
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
);

my $size_widget = 180;

$mw->insert('Widget::Time' =>
    origin  => [(($app_width/2)-($size_widget/2)), 220],
    size    => [$size_widget, 39],
    name    => 'Time',
    time    => [0,0,0],
    format  => 'hh:mm:ss',
    alignment => ta::Center,
    font    => { size => 26 },
    enabled => 0,
    disabledBackColor => cl::LightGray,
    disabledColor      => cl::Black,
);
```

15.2.2 Timer Variables and Progress Bar

```
my $remaining_time = 60;
my $copy_remaining_time;
my $start = 0;

my $progressbar = $mw->insert( 'Prima::ProgressBar',
    origin => [5, 140],
    width  => $app_width-10,
    height => 40,
    max    => $remaining_time,
    min    => 0,
    value  => $remaining_time,
    color  => cl::Green,
);
```

15.2.3 Reset Function

```
sub reset_timer {
    my $stop = shift;
    $mw->mcTimer->stop if $stop;

    $btn->text('Start');
    $start = 0;

    $remaining_time = 60;
    $copy_remaining_time = $remaining_time;

    $duration_combo->text(60);

    $progressbar->color(cl::Green);
    $progressbar->max($remaining_time);
    $progressbar->value($remaining_time);

    $mw->Time->time([0,0,0]);
    $duration_combo->enabled(1);
}
```

15.2.4 Timer Tick Logic (start_timer)

This is the heart of the countdown.

```
sub start_timer {
    if ($start == 0) {
        $progressbar->max($remaining_time);
        $copy_remaining_time = $remaining_time;
        $start = 1;
    }

    $duration_combo->enabled(0);

    $progressbar->color(0xffa500)
        if $remaining_time < 0.5 * $copy_remaining_time;

    $progressbar->color(cl::Red)
        if $remaining_time < 11;

    if ($remaining_time >= 0) {
        my ($h, $m, $s) = (
            int($remaining_time / 3600),
            int(($remaining_time % 3600) / 60),
            int($remaining_time % 60),
        );

        $mw->Time->time([$h,$m,$s]);
        $progressbar->value($remaining_time);
        $remaining_time--;
    }
    else {
        $mw->mcTimer->stop;
        showMessageDialog(300, 150, "Info",
            "Countdown complete!\n\nThe program will now reset all.");
        reset_timer(0);
    }
}
```

15.2.5 Timer Object

```
$mw->insert(Timer =>
    name    => 'mcTimer',
    timeout => 1000,
    onTick  => sub { start_timer() },
);
```

15.2.6 Duration Selector (ComboBox)

```
$duration_combo = $mw->insert(ComboBox =>
    origin => [(($app_width/2) + 100), 70],
    size   => [50, 30],
    items  => [60 .. 300],
    style  => cs::DropDown,
    readOnly => 1,
    onChange => sub { $remaining_time = $_[0]->text },
);
```

15.2.7 Start/Stop Button

```
$btn = $mw->insert(Button =>
    origin => [(($app_width/2)-(100/2)), 15],
    size   => [100, 30],
    text   => 'Start',
    onClick => sub {
        $btn = $_[0];
        if ($btn->text eq 'Start') {
            $btn->text('Stop');
            $mw->mcTimer->start;
        }
        else {
            reset_timer(1);
        }
    },
);
```

15.2.8 Full Program Countdown Timer

```
use Prima qw(Application Buttons Label ComboBox Sliders Widget::Time );

use lib '.';
require "showMessageDialog.pl";

my $app_width = 600;

my $mw = Prima::MainWindow->new(
    text => 'Progress bar',
    size => [ $app_width, 280 ],
    backColor => cl::LightGray,
    icon  => Prima::Icon->load('icon.png'),
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
);

# Define size for the time display widget
my $size_widget = 180;

# Insert a digital clock/time display widget into the main window
$mw->insert( 'Widget::Time' =>

    origin  => [(($app_width/2) - ($size_widget/2)), 220],
    size    => [$size_widget, 39],
    name => 'Time',
    # Initial time 00:00:00
    time => [0,0,0],
    # Time format
    format => 'hh:mm:ss',

    borderWidth => 0,
    alignment => ta::Center,
    font   => { size => 26, },

    enabled => 0,
    # see previous code 15.1
    disabledBackColor => cl::LightGray,
    disabledColor => cl::Black,

);

my $remaining_time = 60;
my $copy_remaining_time;
my $btn;
my $duration_combo;
my $start = 0;

my $progressbar = $mw->insert('Prima::ProgressBar',
    origin => [ 5, 140 ],
    width => $app_width-10,
    height => 40,
    max => $remaining_time,
    min => 0,
    value => $remaining_time,
    color => cl::Green,
);

# Reset function
sub reset_timer {
        my $stop = @_;
        $mw->mcTimer->stop if ($stop);
```

```
    $btn->text('Start');
    $start = 0;
    $remaining_time = 60;
    $copy_remaining_time = $remaining_time;
    $duration_combo->text(60);
    $progressbar->color(cl::Green);
    $progressbar->max($remaining_time);
    $progressbar->min(0);
    $progressbar->value($remaining_time);
    $mw->Time->time([0,0,0]);
    $duration_combo->enabled(1);
}

# Timer tick function
sub start_timer {

    if ($start == 0) {
        $progressbar->max($remaining_time);
        $copy_remaining_time = $remaining_time;
        $start = 1;
    }
    $duration_combo->enabled(0);
    # the first color change after 50% elapsed time
    $progressbar->color(0xffa500) if ($remaining_time <
                                      ( 0.5 * $copy_remaining_time) );
    # always the last 10 seconds color red
    $progressbar->color(cl::Red) if ($remaining_time < 11 );

    if ($remaining_time >= 0) {

        my ($seconds, $minutes, $hours) = (
                                        (int($remaining_time / 3600)),
                                        (int(($remaining_time % 3600) / 60)),
                                        (int($remaining_time % 60)) );

        $mw->Time->time([$hours,$minutes,$seconds]);
        $progressbar->value($remaining_time);
        $remaining_time--;

    }
    else {
        $mw->mcTimer->stop;
        showMessageDialog(
            300, 150,
            "Info",
            "Countdown complete!\n\nThe program will now reset all.");
        reset_timer(0);
    }
}

$mw->insert( Timer =>
        name => 'mcTimer',
        timeout => 1000,
        onTick  => sub {
                start_timer,
        },
);

$mw->insert( Label =>
    origin => [(($app_width/2) - (200/2)), 70],
    size   => [200, 25],
    text   => "Choose duration (seconds)",
    font   => { size => 11, }
```

```
    font => { size => 11, },
    color => cl::Black,
);

# Duration selection ComboBox
$duration_combo = $mw->insert( ComboBox =>
    origin => [(($app_width/2) + 100), 70],
    size => [50, 30],
    name => $remaining_time,
    items => [ 60 .. 300 ],
    style => (cs::DropDown),
    editProfile => { backColor => cl::White, color => cl::Black },
    listProfile => { backColor => cl::White, color => cl::Black },
    readOnly => 1,
    # Update $remaining_time when changed
    onChange => sub{ $remaining_time = $_[0]->text; },
);

# Start/Stop button
$btn = $mw->insert( Button =>
                        origin => [(($app_width/2) - (100/2)), 15],
                        name => 'start_button',
                        size => [100, 30],
                        text => "Start",
                        onClick => sub {
                $btn = $_[0];
                if ($btn->text eq 'Start') {
                    # start or resume
                    $btn->text('Stop');
                    # if it was a fresh start, start_timer() will do
                    # initialization
                    $mw->mcTimer->start;
                } else {
                                        reset_timer(1);
                }
                        },
);

Prima->run;
```

Listing 15.2: Timer Class and Progressbar

Later I'll show you how I built a custom timer in Prima, using the visual builder.

15.3 Toggle Buttons and Interactive State Controls

Many applications need controls that change appearance or behavior depending on the current mode. In Prima, you can combine radio buttons, checkboxes, and toggle-style buttons to build dynamic interfaces that react to user input.

In this section, we build a small demo where:

- Selecting Mode A, B, or C updates a color panel.
- Mode B enables a checkbox.
- Mode C enables a custom toggle button.
- The interface updates immediately depending on the user's choice.

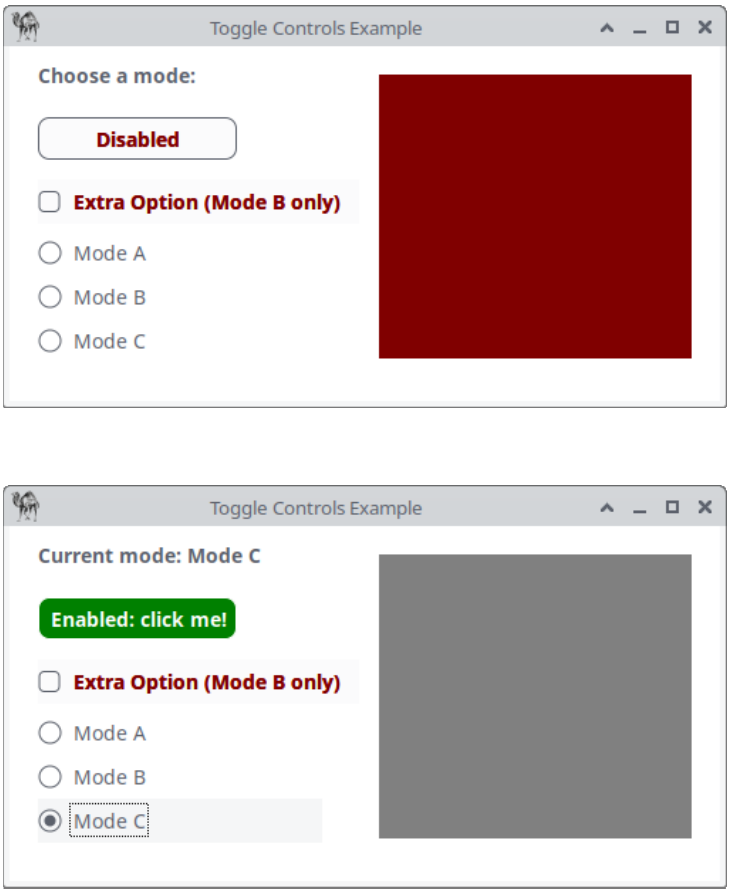


Figure 15.3: Toggle Controls

15.3.1 The Main Window and Panel

We begin by creating the window and a panel that will reflect the current mode (through background color).

```
my $mw = Prima::MainWindow->new(
    text      => 'Toggle Controls Example',
    size      => [500, 250],
    backColor => cl::White,
    icon      => Prima::Icon->load('icon.png'),
);

# Panel reflecting the current mode
my $panel = $mw->insert(Widget =>
    origin    => [260, 30],
    size      => [220, 200],
    backColor => cl::Red,
);
```

The panel is our **visual feedback element**. Every mode will change its color.

15.3.2 Status Label

We display the current mode at the top left. The `Markup` module gives us bold text.

```
my $label = $mw->insert(Label =>
    text   => M 'B<Choose a mode:>',
    origin => [20, 215],
);
```

15.3.3 Toggle Button (Enabled Only in Mode C)

This button changes the panel color between **Cyan** and **Green**, but it is initially disabled.

```
my $toggle_state = 0;

my $btn_toggle = $mw->insert(Button =>
    text      => M 'B<C<Red|Disabled>>',
    origin    => [20, 170],
    size      => [140, 30],
    enabled   => 0,
    backColor => cl::White,
    color     => cl::Black,
    onClick   => sub {
        $toggle_state = !$toggle_state;
        $panel->backColor($toggle_state ? cl::Green : cl::Cyan);
        $label->text(M "B<Custom Toggle: " . ($toggle_state ? 'ON' : 'OFF') . ">");
    },
);
```

Key point: This toggle is *only accessible* when Mode C is selected.

15.3.4 Checkbox (Enabled Only in Mode B)

Mode B activates a checkbox that modifies the panel color.

```
my $chk_extra = $mw->insert(CheckBox =>
    text    => M 'B<C<Red|Extra Option (Mode B only)>>',
    origin  => [20, 125],
    enabled => 0,
    onClick => sub {
        $panel->backColor($_[0]->checked ? cl::Black : cl::Gray);
    },
);
```

When checked, the panel turns black, otherwise gray.

15.3.5 Radio Modes and Dynamic State Switching

The three radio buttons drive the entire interface. Their `onRadioClick` event enables or disables widgets depending on the selected mode.

```
my $grp = $mw->insert(GroupBox =>
    text   => 'Modes',
    origin => [20, 20],
    size   => [200, 100],
    border => 0,
    onRadioClick => sub {
        my $mode = $_[1]->text;
        $label->text(M "B<Current mode: $mode>");

        # Reset UI elements
        $btn_toggle->enabled(0);
        $btn_toggle->text(M 'B<C<Red|Disabled>>');
        $btn_toggle->backColor(cl::White);
        $btn_toggle->color(cl::Black);

        $chk_extra->enabled(0);
        $chk_extra->text(M 'B<C<Red|Extra Option (Mode B only)>>');
        $toggle_state = 0;

        # Mode-specific behavior
        if    ($mode eq 'Mode A') { $panel->backColor(cl::LightBlue); }
        elsif ($mode eq 'Mode B') {
            $panel->backColor(cl::Yellow);
            $chk_extra->enabled(1);
            $chk_extra->text(M 'B<C<Green|Extra Option (Mode B only)>>');
        }
        elsif ($mode eq 'Mode C') {
            $panel->backColor(cl::Gray);
            $btn_toggle->enabled(1);
            $btn_toggle->backColor(cl::Green);
            $btn_toggle->color(cl::White);
            $btn_toggle->text(M 'B<C<White|Enabled: click me!>>');
        }
    },
);
```

Then we insert the radio buttons:

```
$grp->insert(Radio =>
    text => 'Mode A', pack => { side => 'top', fill => 'x' });
$grp->insert(Radio =>
    text => 'Mode B', pack => { side => 'top', fill => 'x' });
$grp->insert(Radio =>
    text => 'Mode C', pack => { side => 'top', fill => 'x' });
```

Finally:

```
Prima->run;
```

15.3.6 Full Program Toggle Controls

```
use Prima qw(Application Buttons Label);
use Prima::Drawable::Markup q(M);

my $mw = Prima::MainWindow->new(
    text => 'Toggle Controls Example',
    size => [500, 250],
    backColor => cl::White,
    icon => Prima::Icon->load('icon.png'),
);

# ----------------------------------------
# Panel that reflects current mode/state
# ----------------------------------------
my $panel = $mw->insert(Widget =>
    origin    => [260, 30],
    size      => [220, 200],
    backColor => cl::Red
);

# ----------------------------------------
# Status label
# ----------------------------------------
my $label = $mw->insert(Label =>
    text   => M 'B<Choose a mode:>',
    origin => [20, 215],
);

# ----------------------------------------
# Toggle Button  (Mode C only)
# ----------------------------------------
my $toggle_state = 0;

my $btn_toggle = $mw->insert(Button =>
    text    => M 'B<C<Red|Disabled>>',
    origin  => [20, 170],
    size    => [140, 30],
    enabled => 0,
    backColor => cl::White,
    color     => cl::Black,

    onClick => sub {
        $toggle_state = !$toggle_state;
        $panel->backColor($toggle_state ? cl::Green : cl::Cyan);
        $label->text(M "B<Custom Toggle: " . ($toggle_state ?
                                                        'ON' : 'OFF') .
                                                        ">");
    },
);

# ----------------------------------------
# Checkbox (Mode B only)
# ----------------------------------------
my $chk_extra = $mw->insert(CheckBox =>
    text    => M 'B<C<Red|Extra Option (Mode B only)>>',
    origin  => [20, 125],
    enabled => 0,

    onClick => sub {
        $panel->backColor($_[0]->checked ? cl::Black : cl::Gray);
    },
);
```

```
# ---------------------------------------
# Radio Modes
# ---------------------------------------
my $grp = $mw->insert(GroupBox =>
    text   => 'Modes',
    origin => [20, 20],
    size   => [200, 100],
    border => 0,

    onRadioClick => sub {
        my $mode = $_[1]->text;
        $label->text(M "B<Current mode: $mode>");

        # Reset controls
        $btn_toggle->enabled(0);
        $btn_toggle->text(M 'B<C<Red|Disabled>>');
        $btn_toggle->backColor(cl::White);
        $btn_toggle->color(cl::Black);

        $chk_extra->enabled(0);
        $chk_extra->text(M 'B<C<Red|Extra Option (Mode B only)>>');

        $toggle_state = 0;

        if ($mode eq 'Mode A') {
            $panel->backColor(cl::LightBlue);
        }
        elsif ($mode eq 'Mode B') {
            $panel->backColor(cl::Yellow);
            $chk_extra->enabled(1);
            $chk_extra->text(M 'B<C<Green|Extra Option (Mode B only)>>');
        }
        elsif ($mode eq 'Mode C') {

            $panel->backColor(cl::Gray);
            $btn_toggle->enabled(1);
            $btn_toggle->backColor(cl::Green);
            $btn_toggle->color(cl::White);
            $btn_toggle->text(M 'B<C<White|Enabled: click me!>>');
        }
    },
);

$grp->insert(Radio =>
    text => 'Mode A', pack => { side => 'top', fill => 'x' });
$grp->insert(Radio =>
    text => 'Mode B', pack => { side => 'top', fill => 'x' });
$grp->insert(Radio =>
    text => 'Mode C', pack => { side => 'top', fill => 'x' });

Prima->run;
```

Listing 15.3: Toggle Controls

Closing Words

In this part, you learned how to bring your Prima applications to life with **dynamic**, **time-based**, and **interactive** widgets. We explored timers, status displays, progress indicators, and user-controlled interface states. By breaking each program into clear conceptual chunks, the underlying mechanics become easier to understand, while the complete listings provide runnable reference implementations.

Each section includes small code chunks for clarity, followed by a complete annotated program in its own full listing. These listings allow you to study

or extend each example in context.

Listing 15.1 Stopwatch
A time-tracking application demonstrating `Widget::Time`, Prima's `Timer` class, and a simple state machine for Start/Stop/Pause logic.

Listing 15.2 Countdown Timer
A countdown tool with a progress bar, dynamic color feedback, a duration selector, and a Start/Stop toggle button.

Listing 15.3 Toggle Controls Example
An interface showcasing radio buttons, checkboxes, and a custom toggle button, illustrating how to manage interactive application state.

Part 6 - Layout and Interface Organization

16. Organizing Widgets Spatially

Chapter 16 presents tools for arranging widgets within an application's interface:

- 16.1 Using Containers and Pack Introduces the use of container widgets and the Pack geometry manager to control basic spatial layout.
- 16.2 Static Multi-Pane Layouts Describes methods for creating fixed multi-pane interfaces that remain consistent in structure.
- 16.3 Dynamic Panes with FrameSet Explains how FrameSet enables resizable and adjustable pane layouts for more flexible interfaces.

16.1 Using Containers and Pack

You can use the `origin` property to position widgets within the `MainWindow`. However, as we've seen, a simpler alternative is the pack geometry manager, which arranges widgets in a container using different orientations (such as top, bottom, left, or right).

Let's experiment with `pack`. Here's an example of a horizontally centered `Label`(`pack` centers it horizontally):

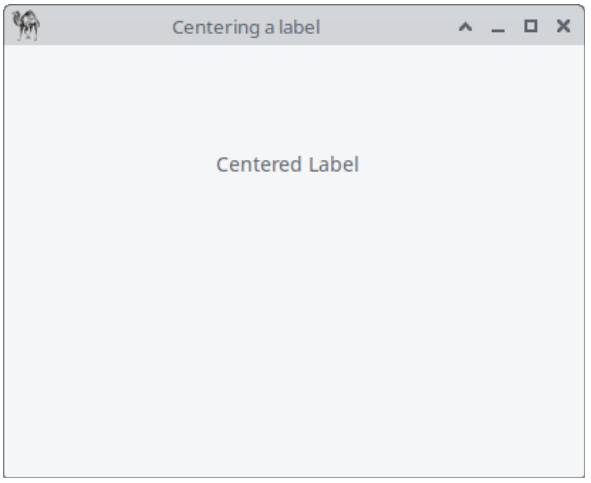


Figure 16.1: A Horizontal Centered Label

```
use Prima qw(Application Label);

my $mw = Prima::MainWindow->new(
    text => 'Centering a label',
    size => [400, 300],
    icon => Prima::Icon->load('icon.png'),
);

my $label = $mw->insert(Label =>
    text => 'Centered Label',
    size => [110, 20],
    pack => {
        side   => 'top',
        # Increased padding to move the label up, decrease to move down
        pady   => 150,
    },
);

Prima->run;
```

Listing 16.1: A Horizontal Centered Label

When you resize the window, the `Label` stays horizontally centered.

Let's add two buttons at the bottom. Notice that we define a container for the buttons!

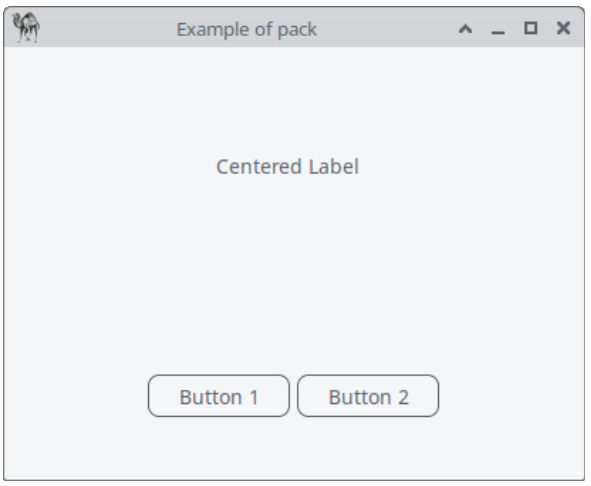


Figure 16.2: A Horizontal Centered Label and Buttons

```
use Prima qw(Application Label Buttons);
my $mw = Prima::MainWindow->new(
    text => 'Example of pack',
    size => [400, 300],
    icon => Prima::Icon->load('icon.png'),
);
my $label = $mw->insert( Label =>
    text => 'Centered Label',
    size => [110, 20],
    # Increased padding to move the label up, decrease to move down
    pack => { side => 'top', pady => 150 },
);

# Creating a widget as a container for buttons
my $button_container = $mw->insert( Widget =>
    # Increased padding to move the button container up, decrease to move down
    pack => { side => 'bottom', pady => 80 },
);
# Adding two buttons next to each other in the button container
my $button1 = $button_container->insert( Button =>
    text => 'Button 1',
    size => [100, 30],
    pack => { side => 'left', padx => 5 },
);
my $button2 = $button_container->insert( Button =>
    text => 'Button 2',
    size => [100, 30],
    pack => { side => 'left', padx => 5 },
);

Prima->run;
```

Listing 16.2: A Horizontal Centered Label and Buttons

When you resize the window, the `Label` and `Widget` stay horizontally centered.

Remember these `pack` tips

Tip	How to Do It	Example
Stack vertically	`pack => { side => 'top' }`	Labels, buttons in a column
Place horizontally	`pack => { side => 'left' }`	Buttons in a row
Mix directions	Use containers for different sections	Label on top, buttons at bottom
Add padding	`pack => { padx => 10, pady => 10 }`	Extra space around widgets
Center widgets	`pack => { side => 'top' }`	Centered labels or buttons

Table 16.1 pack tips

Using `pack` efficiently means keeping your layouts simple, using directions and padding wisely, and organizing widgets into logical sections. But I've to say I still use `origin`. Why and when?

- **Pixel-perfect placement**: when you need a widget at an exact `(x, y)` position (e.g., a custom-drawn background or a fixed overlay).
- **Complex or static designs**: if your layout doesn't need to adapt to resizing, `origin` gives you full control.

But if you need responsive designs, e.g. if your window resizes, `pack` can automatically rearrange widgets, while `origin` requires manual recalculation. So a rule of thumb: use `pack` for simple, flexible layouts.

16.2 Static Multi- Pane Layouts

The pack layout manager is a simpler and more traditional tool for arranging widgets within their parent container (e.g., window) according to packing options like top, left, right, or bottom. If your layout doesn't require resizable sections and you just need to organize widgets in a straightforward, stacked, or aligned manner - either vertically or horizontally- pack is easy to use and works well for this purpose.

Figure 16.3: Multipane Interface

```
use Prima qw(Application Label ComboBox );

my %colors = (
   'black' => 0x000000,
   'white' => 0xFFFFFF,
   'maroon' => 0x800000,
   'red' => 0xFF0000,
   'purple' => 0x800080,
   'green' => 0x008000,
);

my $mw = Prima::MainWindow->new(
    text   => "Static multi-pane interface",
    size => [600,300],
    icon => Prima::Icon->load('icon.png'),
);

my $g1 = $mw->insert( Widget =>
    pack => { side => 'left', fill => 'both', expand => 1 },
    size => [$mw->width/2, $mw->height],
    backColor => 0x000000,
);

my $g2 = $mw->insert( Widget =>
    pack => { side => 'left', fill => 'both', expand => 1 },
    size => [$mw->width/2, $mw->height],
    backColor => cl::White,
);

$g1->insert( Label =>
    pack => { fill => 'none', side => 'top', pady => 15 },
    text   => "Color picker",
    alignment => ta::Center,
    font   => { size => 14, },
    color => cl::White,
);

$g1->insert( ComboBox =>

    # fill => 'none': do not stretch the ComboBox; keep its natural width.
    # side => 'top': put it at the top of the container $g1.
    # pady => 15: add 15 pixels of vertical space above and below the widget.
    pack => { fill => 'none', side => 'top', pady => 15 },
    size => [200, 30],
    items => [ 'black', 'white', 'maroon', 'red', 'purple', 'green' ],
    # when the program starts, "white" is shown by default.
    text  => 'white',
    # cs::DropDown means it shows a dropdown list and lets the user type
    # their own text if needed.
    style => (cs::DropDown),
    onSelectItem => sub {
        # my ($cb) = @_; $cb is the ComboBox object itself.
            my ($cb) = @_;
        # $cb->focusedItem:  gets the index of the selected item in the
        # items array.
        my $idx = $cb->focusedItem;
        # $cb->items->[$idx]: converts the index to the actual string name
        # (e.g., "red").
        # return unless defined $idx && $idx >= 0;
        my $name = $cb->items->[$idx];
        setColorFrame($name);
```

```
    },
);

# setColorFrame is a helper function that changes the background color of
# $g2 (the right-side container).
sub setColorFrame {
    my ($name) = @_;
    return unless defined $name && exists $colors{$name};
    # $g2->backColor( $colors{$name} ); sets $g2's background to the
    # corresponding hex color from %colors.
    $g2->backColor( $colors{$name} );
}

Prima->run;
```

Listing 16.3: Multipane Interface

16.3 Dynamic Panes with FrameSet

As already said, if your UI needs resizable sections where users can dynamically adjust the layout by dragging dividers (splitters) between frames, you should use the `FrameSet` class. Two examples:

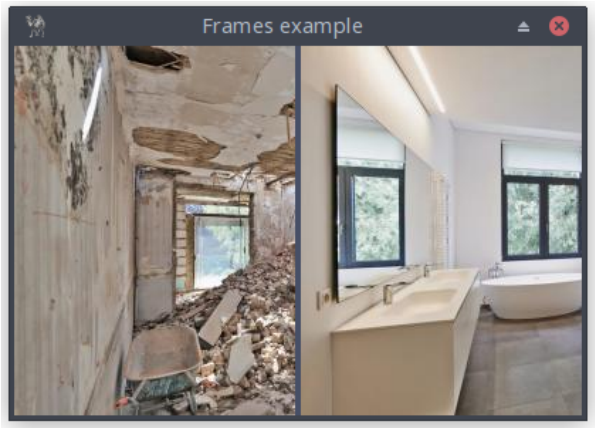
16.3.1 FrameSet, Label and Widget

Figure 16.4: Dynamic Multipane Interface

```
use Prima qw( Label FrameSet Application );

my $mw = Prima::MainWindow->new(
    text   => "Frames example",
    size => [575, 250],
    icon => Prima::Icon->load('icon.png'),
    borderIcons => bi::SystemMenu,
);

my $frame = $mw->insert( FrameSet =>
    size => [$mw->size],
    origin => [0, 0],
    frameSizes => [qw(1% *)],
    flexible => 1,
    # 4 is default
    sliderWidth => 4,
);

$frame->insert_to_frame(
    0,
    Label =>
        pack => { expand => 1, fill => 'both', },
        text => "   \n\n   Whether you think that you can, or that you " .
                "can't, you are usually right.\n\n   Henry Ford",
        color => 0x800000,
        backColor => 0x808080,
);

$frame->insert_to_frame(
    1,
    Widget =>
        pack => { expand => 1, fill => 'both' },
        backColor => 0xFFFF00,
);

Prima->run;
```

Listing 16.4: Dynamic Multipane Interface

16.3.2 FrameSet and ImageViewer



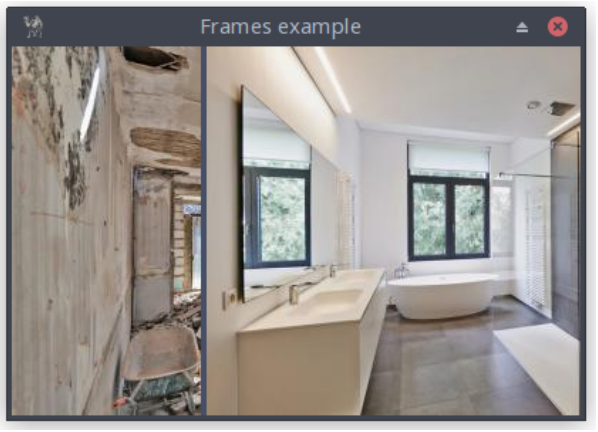


Figure 16.5: Dynamic Multipane Interface

```
use Prima qw( ImageViewer FrameSet Application );

# Load two images that will be displayed side by side.
my $image_left  = Prima::Image->load('Room-Construction-Before2.jpg')
    or die "$@" unless $image_left;

my $image_right = Prima::Image->load('Room-Construction-After2.jpg')
    or die "$@" unless $image_right;

# Create the main window sized to the first image.
my $mw = Prima::MainWindow->new(
    text        => "Frames example",
    width       => $image_left->width,
    height      => $image_left->height,
    icon        => Prima::Icon->load('icon.png'),
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu | bi::TitleBar,
);

# ------------------------------------------------------------------
# FrameSet: a container that splits its area into adjustable panes.
#
# - frameSizes => [ '50%', '*' ] defines two frames:
#       Frame 0 = 50% of the width
#       Frame 1 = remaining space (“*”)
# - flexible => 1 allows the user to drag the separator.
# - separatorWidth => 1 sets a thin dividing line.
#
# This creates a vertical split with two resizable panels.
# ------------------------------------------------------------------
my $frame = $mw->insert( FrameSet =>
    size           => [$mw->size],
    origin         => [0, 0],
    frameSizes     => [qw(50% *)],
    flexible       => 1,
    separatorWidth => 1,
);

# Insert an ImageViewer into the left frame (frame index 0).
# pack => { expand => 1, fill => 'both' } ensures the viewer
# fills its entire pane and resizes with it.
$frame->insert_to_frame(
    0,
    ImageViewer =>
        pack    => { expand => 1, fill => 'both' },
        image   => $image_left,
        hScroll => 0,
        vScroll => 0,
);

# Insert the second image into the right frame (frame index 1).
$frame->insert_to_frame(
    1,
    ImageViewer =>
        pack    => { expand => 1, fill => 'both' },
        image   => $image_right,
        hScroll => 0,
        vScroll => 0,
);

Prima->run;
```

Listing 16.5: Dynamic Multipane Interface

17. Structuring Your UI Logically

Chapter 17 presents three interface components that support clearer layout and organization:

- 17.1 **GroupBox**: Visual Grouping
  Introduces the GroupBox widget, used to visually group related controls within a bordered frame and optional title.
- 17.2 **Notebook**: Tabs for Complex Interfaces
  Presents the Notebook widget, which provides tabbed pages for organizing content in more complex interfaces.
- 17.3 **Panel**: Custom Borders and Backgrounds
  Covers the Panel widget, a flexible container that supports custom borders, backgrounds, and visual adjustments.

17.1 GroupBox: Visual Grouping

This Perl program illustrates the use of containers and layout management in the Prima GUI toolkit. It shows how to organize widgets inside GroupBoxes, use nested containers for spacing and alignment, and control layout with the *pack* geometry manager.

The interface includes an Appearance GroupBox with exclusive radio buttons for theme selection, checkboxes for additional options, and a nested Advanced GroupBox to demonstrate horizontal alignment and vertical centering of child widgets. A status label dynamically displays the current selections, providing feedback while highlighting how widgets are packed and arranged within different containers.

This program serves as a practical example of how to structure a GUI with multiple levels of containers while maintaining clean layout and consistent spacing.

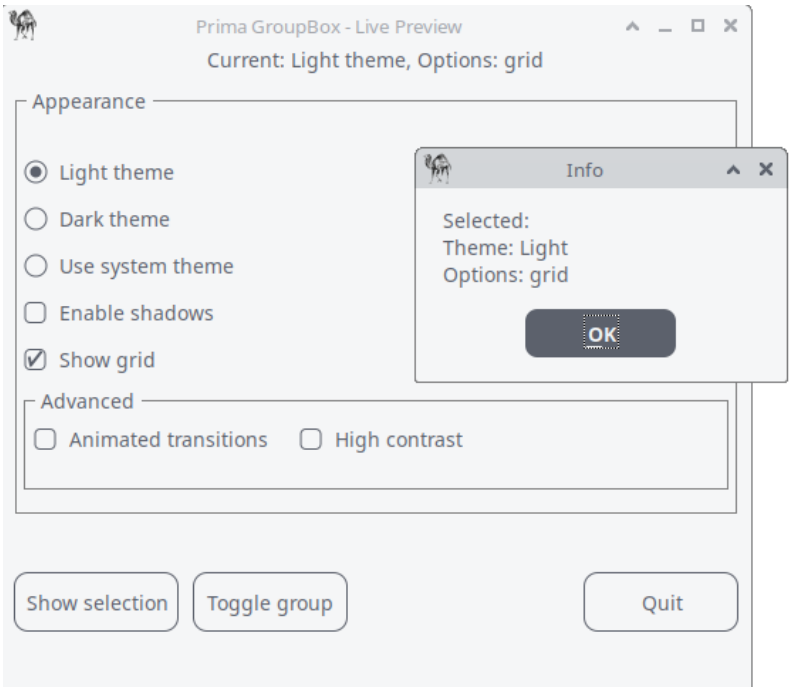


Figure 17.1: Groupbox

```
use Prima qw(Application Buttons Label MsgBox);

# 1. Main Window
my $w = Prima::MainWindow->create(
    text     => 'Prima GroupBox - Live Preview',
    size     => [520, 450],
    centered => 1,
    icon        => Prima::Icon->load('icon.png'),
);

# 2. Preview Label
my $preview = $w->insert(Label =>
    text      => "Current: Waiting for input...",
    pack      => { side => 'top', fill => 'x', padx => 8, pady => 4 },
    alignment => ta::Center,
);

# 3. GroupBox: Appearance
my $gb = $w->insert(GroupBox =>
    text   => 'Appearance',
    pack   => { side => 'top', fill => 'x', padx => 12, pady => 6 },
    height => 300,
);

# spacer to move content below title
$gb->insert(Label => text => '', pack => { side => 'top', fill => 'none',
                                                          pady => 20 });

my %widgets;

# 4. Theme Radio Buttons (exclusive)
$widgets{light} = $gb->insert('Prima::Radio' => (
    text    => 'Light theme',
    pack    => { side => 'top', fill => 'x', padx => 12, pady => 2 },
    group   => 1,  # start radio group
    checked => 1,
    onClick => sub { update_preview($preview, \%widgets); },
));

$widgets{dark} = $gb->insert('Prima::Radio' => (
    text    => 'Dark theme',
    pack    => { side => 'top', fill => 'x', padx => 12, pady => 2 },
    onClick => sub { update_preview($preview, \%widgets); },
));

$widgets{system} = $gb->insert('Prima::Radio' => (
    text    => 'Use system theme',
    pack    => { side => 'top', fill => 'x', padx => 12, pady => 2 },
    onClick => sub { update_preview($preview, \%widgets); },
));

# 5. Checkboxes
$widgets{shadows} = $gb->insert('Prima::CheckBox' => (
    text    => 'Enable shadows',
    pack    => { side => 'top', fill => 'x', padx => 12, pady => 2 },
    onClick => sub { update_preview($preview, \%widgets); },
));

$widgets{grid} = $gb->insert('Prima::CheckBox' => (
    text    => 'Show grid',
    pack    => { side => 'top', fill => 'x', padx => 12, pady => 2 },
```

```
    checked => 1,
    onClick => sub { update_preview($preview, \%widgets); },
));

# 6. Nested GroupBox (Advanced)
my $advanced_gb = $gb->insert(GroupBox =>
    text => 'Advanced',
    pack => { side => 'top', fill => 'x', padx => 12, pady => 6 },
);

# small inner row for padding and centering
my $adv_row = $advanced_gb->insert(Widget =>
    pack => { side => 'top', fill => 'x', padx => 6, pady => 40 },
    #height => 28,
);

$widgets{animated} = $adv_row->insert('Prima::CheckBox' => (
    text    => 'Animated transitions',
    pack    => { side => 'left', padx => 8, pady => 1 },
    onClick => sub { update_preview($preview, \%widgets); },
));

$widgets{high_contrast} = $adv_row->insert('Prima::CheckBox' => (
    text    => 'High contrast',
    pack    => { side => 'left', padx => 8, pady => 1 },
    onClick => sub { update_preview($preview, \%widgets); },
));

# 7. Buttons
$w->insert(Button =>
    text    => 'Show selection',
    pack    => { side => 'left', padx => 10, pady => 10 },
    onClick => sub {
        my $theme = $widgets{light}->checked ? 'Light'
                  : $widgets{dark}->checked  ? 'Dark'
                  :                            'System';
        my @opts;
        push @opts, 'shadows'      if $widgets{shadows}->checked;
        push @opts, 'grid'         if $widgets{grid}->checked;
        push @opts, 'animated'     if $widgets{animated}->checked;
        push @opts, 'high contrast' if $widgets{high_contrast}->checked;
        my $opts = @opts ? join(', ', @opts) : 'none';
        message_box("Info", "Selected:\nTheme: $theme\nOptions: $opts",
                            mb::OK, compact => 1,);
    },
);

$w->insert(Button =>
    text    => 'Toggle group',
    pack    => { side => 'left', padx => 10, pady => 10 },
    onClick => sub {
        my $new = !$gb->enabled;
        $gb->enabled($new);
        $gb->text($new ? 'Appearance' : 'Appearance (disabled)');
    },
);

$w->insert(Button =>
    text    => 'Quit',
    pack    => { side => 'right', padx => 10, pady => 10 },
    onClick => sub { exit },
);
```

```
# 8. Update Preview Subroutine
sub update_preview {
    my ($preview_label, $widgets) = @_;

    # Determine theme
    my $theme = $widgets->{light}->checked ? 'Light'
              : $widgets->{dark}->checked  ? 'Dark'
              :                              'System';

    # Collect options
    my @opts;
    push @opts, 'shadows'      if $widgets->{shadows}->checked;
    push @opts, 'grid'         if $widgets->{grid}->checked;
    push @opts, 'animated'     if $widgets->{animated}->checked;
    push @opts, 'high contrast' if $widgets->{high_contrast}->checked;
    my $opts = @opts ? join(', ', @opts) : 'none';

    # Update preview
    $preview_label->text("Current: $theme theme, Options: $opts");
}

# 9. Initialize Preview
update_preview($preview, \%widgets);

Prima->run;
```

Listing 17.1: Groupbox

17.2 Notebook - Tabs for Complex Interfaces

This Perl script showcases a tabbed interface designed to explore educational content about the Solar System. Its features:

It uses the `Prima::TabbedNotebook` widget to organize content into three tabs: Intro, Earth, and Quiz.

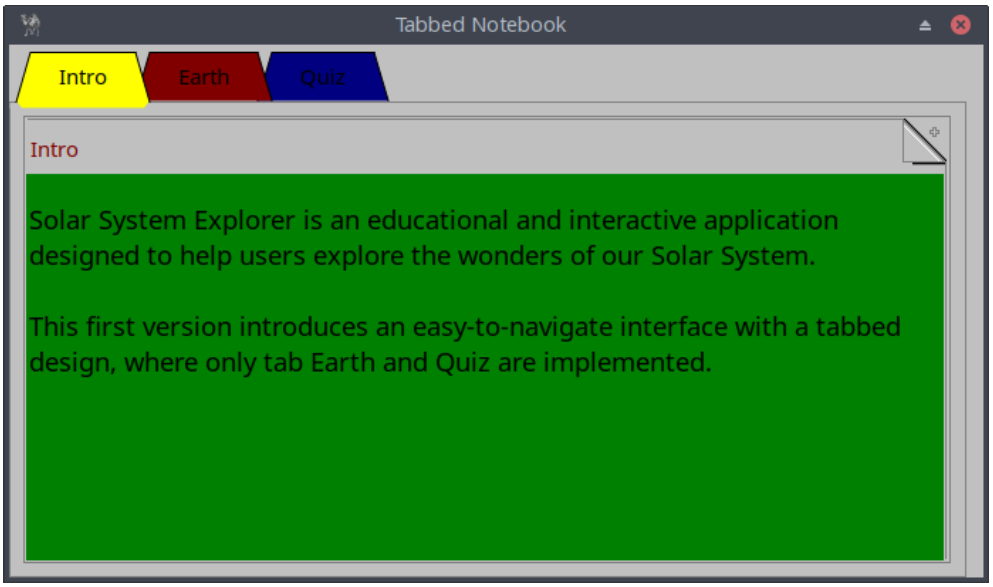


Figure 17.2: A Colored, Tabbed Notebook With Three Tabs

```
my $nb = $w->insert('Prima::TabbedNotebook' =>
    pack => {side => 'left',},
    size => [ 590, 300 ],
    color => cl::Red,
    tabs => [ 'Intro', 'Earth', 'Quiz', ],
    # settings of textcolor and backgroundcolor of notebook object,
    # part of TabbedNotebook
    notebookProfile => {color => cl::Black, backColor => cl::Green,},

    # default settings of the TABS
    tabsetProfile => {color => cl::Black, backColor => cl::LightGray},

    # each TAB has its own color
    colored => 1,
    colorset => [cl::Yellow, cl::Red, cl::Blue],
);
```

Each TAB corresponds with a page. To populate the first page with a Label object, write:

```
# define the Label properties
my %intro = (
    origin => [2, 100],
    text => ($str1 . "\n\n" . $str2),
    autoHeight => 1,
    width => 550,
    wordWrap => 1,
    color => cl::Black,
    font => { size => 14, },
);
# insert_to_page INDEX, CLASS, %PROFILE, [[ CLASS, %PROFILE], ... ]
$nb->insert_to_page( 0, 'Prima::Label', %intro, ),
```

The page EARTH defines *$degreeCelcius = "\x{00B0}", ``which* is a way to define a string in Perl that represents the degree symbol (°) using its Unicode code point: *\x{00B0}* refers to the Unicode character *U+00B0*, which is the degree symbol (°).

The page EARTH also show how to make a hyperlink to a webpage.

The TAB Quiz offers an interactive quiz with multiple-choice questions about Earth. Users can answer questions, receive feedback, and navigate through a series of questions using a "Next question!" button.

$nb->insert_to_page(2, insert_answers);

is the code to populate the third page, where ‘insert_antwers’ is a method that generates four times array references *CLASS*, *%PROFILE*.

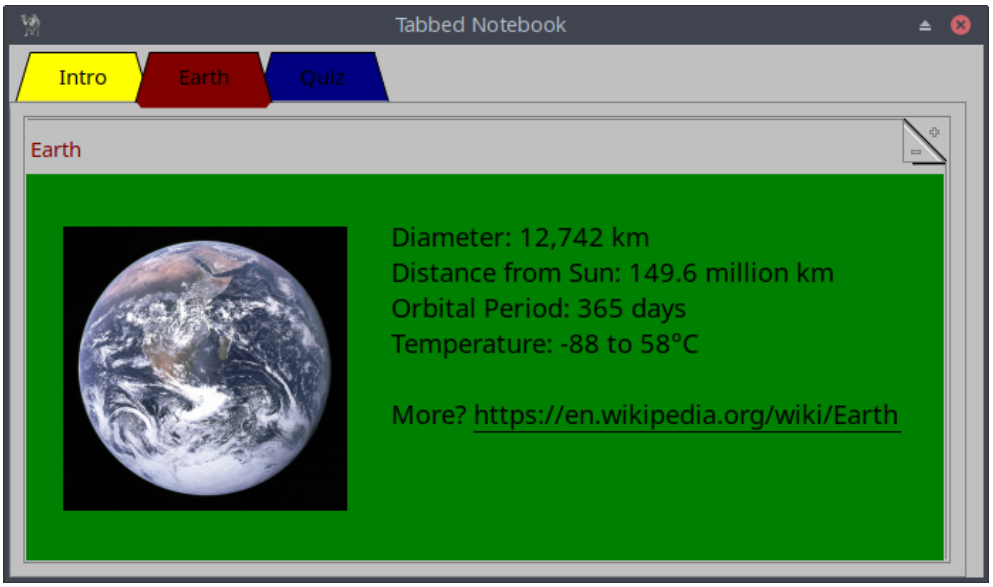


Figure 17.3: Tab Earth

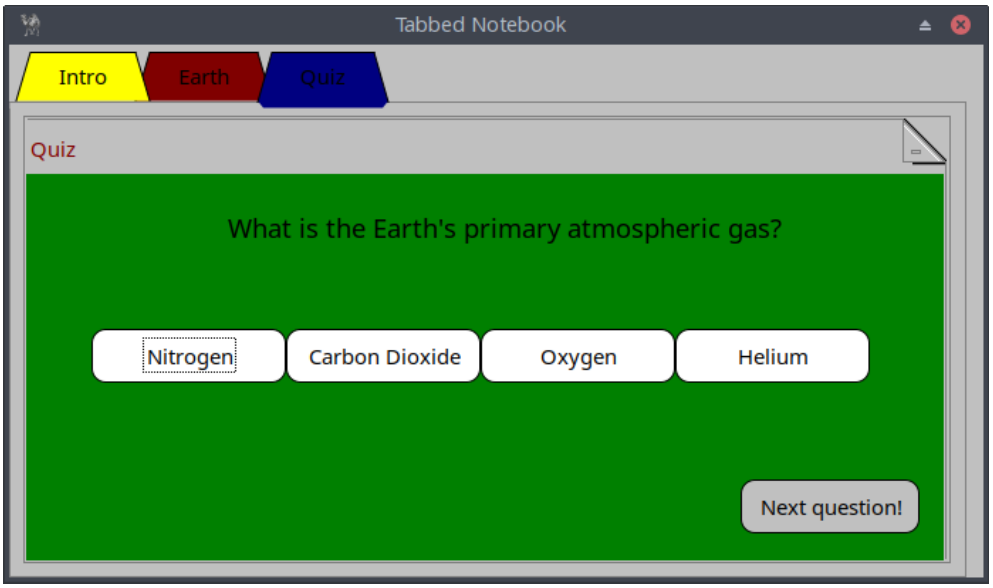


Figure 17.4: Tab Quiz

```
use Prima qw(Notebooks Buttons Label ImageViewer Application);
use Prima::Drawable::Markup q(M);
use lib '.';
require "showMessageDialog.pl";
my $w = Prima::MainWindow->new(
    name => 'Tabbed Notebook',
    size => [ 600, 300],
    designScale => [7, 16],
    backColor => cl::LightGray,
    color => cl::Black,
    icon  => Prima::Icon->load('icon.png'),
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
);

my $nb = $w->insert('Prima::TabbedNotebook' =>
    pack => {side => 'left',},
    size => [ 590, 300 ],
    color => cl::Red,
    tabs => [ 'Intro', 'Earth', 'Quiz', ],
    tabsetProfile => {color => cl::Black, backColor => cl::LightGray},
    notebookProfile => {color => cl::Black, backColor => cl::Green,},
    colored => 1,
    colorset => [cl::Yellow, cl::Red, cl::Blue],
);

#-----------------------------------------------------------------------
#first tab Solar System Explorer

my $str1 = "Solar System Explorer is an educational and interactive " .
           "application designed to help users explore the wonders " .
           "of our Solar System.";
my $str2 = "This first version introduces an easy-to-navigate interface " .
           "with a tabbed design, where only tab Earth and Quiz are implemented.";

my %intro = (

    origin => [2, 100],
    text => ($str1 . "\n\n" . $str2),
    autoHeight => 1,

    width => 550,
    wordWrap => 1,

    color => cl::Black,
    font => { size => 14, },
);

$nb->insert_to_page( 0, 'Prima::Label', %intro, ),

#-----------------------------------------------------------------------#second tab Earth

my $mc_image = Prima::Image->load('earth.png') or die "Cannot load image";

my %earth_image = (
    #pack => { side => 'left', fill => 'both' },
    origin => [10, 8],
    size => [200, 200],
    image => $mc_image,
    valignment => ta::Center,
    alignment => ta::Center,
```

```
    backColor => cl::Green,
);

# my $degreeCelcius = "\x{00B0}";

my %key_facts = (

    origin => [225, 70],
    linkColor => cl::Black,
    text   => \ "Diameter: 12,742 km\nDistance from Sun: 149.6 million km\n" .
                "Orbital Period: 365 days\nTemperature: -88 to 58\x{00B0}C\n\nMore?" . "L<https://en.w
    autoHeight => 1,
    font   => { size => 14, },
    color => cl::Black,
);

$nb->insert_to_page( 1, ['Prima::ImageViewer',%earth_image],
                        ['Prima::Label', %key_facts], ),

#------------------------------------------------------------------------
#third tab QUIZ

my @quiz = (

{
    question => "How long does Earth take to orbit the Sun?",
    choices  => ["225 days", "730 days", "365 days", "88 days"],
    answer   => "365 days",
},
{
    question => "What percentage of Earth's surface is covered in water?",
    choices  => ["30%", "50%", "90%", "70%"],
    answer   => "70%",
},
{
    question => "What is the Earth's diameter?",
    choices  => ["15,000 km", "12,742 km", "20,000 km", "10,000 km"],
    answer   => "12,742 km",
},
{
    question => "What is the Earth's primary atmospheric gas?",
    choices  => ["Nitrogen", "Carbon Dioxide", "Oxygen", "Helium"],
    answer   => "Nitrogen",
},
{
    question => "What is the average distance of Earth from the Sun?",
    choices  => ["108.2 million km", "57.91 million km", "149.6 million km",
                 "778.5 million km"],
    answer   => "149.6 million km",
},
{
    question => "What is Earth's only natural satellite?",
    choices  => ["Moon", "Io", "Europa", "Phobos"],
    answer   => "Moon",
},
{
    question => "What is the shape of Earth's orbit around the Sun?",
    choices  => ["Circular", "Oval", "Triangular", "Elliptical"],
    answer   => "Elliptical",
},
{
    question => "What is the thickest layer of Earth?",
    choices  => ["Inner Core", "Crust", "Outer Core", "Mantle"],
```

```
    choices  => ["Inner core", "Crust", "Outer core", "Mantle"],
    answer   => "Mantle",
},
{
    question => "Which zone of Earth supports all life?",
    choices  => ["Atmosphere", "Biosphere", "Hydrosphere", "Lithosphere"],
    answer   => "Biosphere",
},
{
    question => "What causes Earth's seasons?",
    choices  => ["Gravity", "Rotation", "Axis tilt", "Sun distance"],
    answer   => "Axis tilt",
},

);

my $no_questions = scalar(@quiz);

#my $rand = int(rand (scalar @quiz));

my $index = 0;
my %q;
sub insert_question {

    %q = (
                origin => [0, 175],
                text => $quiz[$index]->{question},
                size => [590, 60],
                alignment => ta::Center,
                autoHeight => 1,
                font => { size => 14, },
                color => cl::Black,
    );

}
sub evaluate {
            my ($i, $index) = @_;
            if ($quiz[$index]->{choices}->[$i] eq $quiz[$index]->{answer}) {
                showMessageDialog(250, 125, "Your answer is", "Correct!" );
            } else {
                showMessageDialog(250, 125, "Your answer is", "Wrong..." );
            }
}


$nb->insert_to_page( 2, 'Prima::Label', insert_question );

sub insert_answers {
    my $button_width = 120;
    my $x_position = 40;
    my $y_position = 100;
    my $space_between = 120;

    my @buttons;

    # Loop through choices to create the four buttons
    for my $i (0..3) {
        push @buttons, [
            Button =>
                origin     => [$x_position + $i * $space_between, $y_position],
                size       => [$button_width, 30],
                text       => $quiz[$index]->{choices}->[$i],
                backColor  => cl::White,
```

```
                color      => cl::Black,
                onClick    => sub { evaluate($i, $index); },
        ];
    }

    return @buttons;
}

$nb->insert_to_page( 2, insert_answers );

my %next_question_button = (

    origin => [440, 15],
    size => [110, 30],
    name => "Next question!",
    backColor => cl::LightGray,
    color => cl::Black,

    onClick => sub {

                        ($index < $no_questions -1) ? $index++ : ($index = 0);
                        $nb->insert_to_page( 2,  'Prima::Label',  insert_question );
                        $nb->insert_to_page( 2, insert_answers );
    },

);

$nb->insert_to_page( 2,  'Prima::Button', %next_question_button );

Prima->run;
```

Listing 17.2: A Colored, Tabbed Notebook With Three Tabs

17.3 Panel: Custom Borders and Backgrounds

The Panel widget (`Prima::Widget::Panel`) is a simple yet powerful container used to improve both the *organization* and the *appearance* of a user interface. Like `Widget`, it can contain other widgets, but it adds support for borders, 3D shading, and backgrounds—either a solid color or an image. This makes it ideal for creating visually distinct areas within a window.

Panels are particularly useful when you want to:

1. **Group related controls visually.**
   A panel can hold several widgets (buttons, labels, sliders, etc.) that belong together, such as audio settings or file options.
2. **Decorate or highlight a region.**
   Panels can display colored or image-based backgrounds, helping to separate content areas and improve clarity.
3. **Add visual hierarchy.**
   By adjusting the `raise` and `borderWidth` properties, you can make a panel appear raised or sunken, providing visual depth.
4. **Build reusable interface sections.**
   Panels can serve as self-contained units—such as toolbars, sidebars, or info boxes—that can be easily inserted into different parts of a program.

Main Properties

Property	Type	Description	Default
`borderWidth`	*INTEGER*	Width of the 3D-shade border around the widget.	1
`image`	*OBJECT*	Image object drawn as a tiled background. If	`undef`

Property	Type	Description	Default
		`undef`, the background is filled with `backColor`.	
`imageFile`	*PATH*	Image file to be loaded and displayed. Rarely used, since it does not return a success flag.	`undef`
`raise`	*BOOLEAN*	Border style: `1` = raised, `0` = sunken.	1
`zoom`	*INTEGER*	Zoom level for image display. Acceptable range: 1 – 10.	1

Table 17.1: main properties of the Panel widget

17.3.1 Example 1: A Simple Decorative Panel

This creates a light blue, raised panel that contains a centered label. It visually groups content and provides a framed section of the interface.

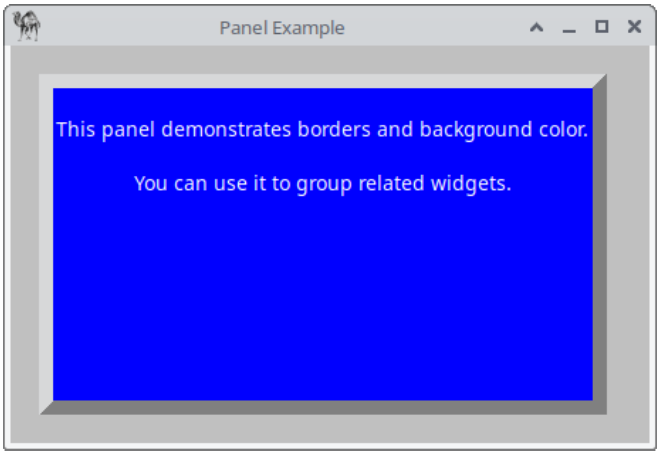


Figure 17.5: A Simple Decorative Panel

```
use Prima qw(Application Label Buttons);
use Prima::Widget::Panel;

my $mw = Prima::MainWindow->new(
    text => 'Panel Example',
    size => [450, 280],
    backColor => cl::LightGray,
    icon => Prima::Icon->load('icon.png'),
);

my $panel = $mw->insert( 'Prima::Widget::Panel',
    origin      => [20, 20],
    size        => [400, 240],
    borderWidth => 10,
    raise       => 1,
    backColor   => cl::LightBlue,
    text        => 'Information Panel',
);

$panel->insert( Label =>
      origin    => [10, 10],
      size      => [380, 200],
      text      => "This panel demonstrates borders and background color.\n\n"
                   .
                   "You can use it to group related widgets.",
      alignment => ta::Center,
      color     => cl::White,
);

Prima->run;
```

Listing 17.3: A Simple Decorative Panel

17.3.2 Example 2 – Reusable Panels (Toolbar and Sidebar)

This example uses two panels to define **reusable layout regions**:

- A **toolbar** (raised) containing buttons.
- A **sidebar** (sunken) for information.

Both panels can be copied or subclassed for use in other windows, showing how the panel serves as a modular building block in interface design.

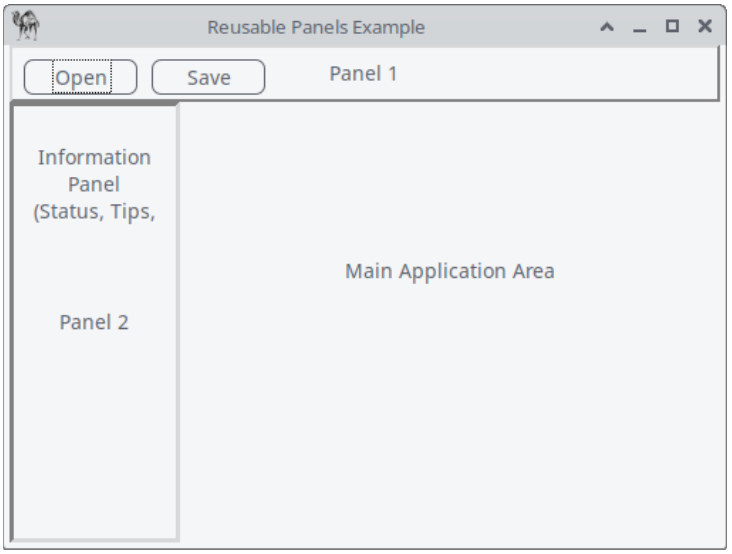


Figure 17.6: Reusable Panels

```
use Prima qw(Application Buttons Label);
use Prima::Widget::Panel;

my $mw = Prima::MainWindow->new(
    text => 'Reusable Panels Example',
    size => [500, 350],
    icon => Prima::Icon->load('icon.png'),
);

# Toolbar panel
my $toolbar = $mw->insert( 'Prima::Widget::Panel',
    origin      => [0, 310],
    size        => [500, 40],
    name => 'Panel 1',
    borderWidth => 2,
    raise       => 1,
);

$toolbar->insert(
    Button => text => 'Open', origin => [10, 5], size => [80, 25],
);
$toolbar->insert(
    Button => text => 'Save', origin => [100, 5], size => [80, 25],
);

# Sidebar panel
my $sidebar = $mw->insert( 'Prima::Widget::Panel',
    name => 'Panel 2',
    origin      => [0, 0],
    size        => [120, 310],
    borderWidth => 3,
    raise       => 0,
);

$sidebar->insert( Label =>
    text      => "Information Panel\n(Status, Tips, etc.)",
    origin    => [10, 220],
    size      => [100, 60],
    wordWrap  => 1,
    alignment => ta::Center,
);

$mw->insert( Label =>
    text      => "Main Application Area",
    origin    => [140, 140],
    size      => [340, 60],
    alignment => ta::Center,
    wordWrap  => 1,
);

Prima->run;
```

Listing 17.4: Reusable Panels

Of course, you could use the **backColor** property to make the panels more discernible.

17.3.3 Do You Really Need a Panel?

`Panels` enhance appearance, but not every layout benefits from them. In many cases, a simple widget - such as a `Label`, `ListBox`, or `Edit` component- can provide the same functionality without the extra layer of containment.

17.3.3.1 A Status Bar

This application demonstrates several interactive elements—such as a button, combo box, checkbox, and spin edit input—each with a designated status bar, implemented as a `Label` class, for feedback.

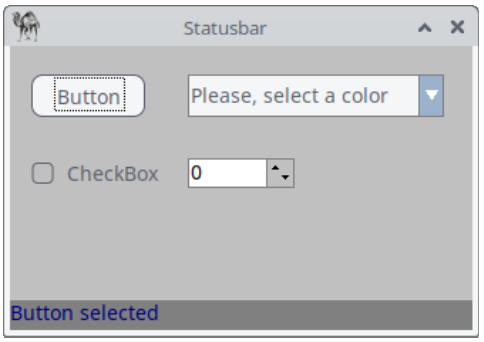


Figure 17.7: A Status Bar

```
use Prima qw(Application Label Buttons ComboBox Sliders);

my $mw = Prima::MainWindow->new(
    text => 'Statusbar',
    size => [325, 200],
    backColor => cl::LightGray,
    icon => Prima::Icon->load('icon.png'),
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
);

my $statusbar;

$mw->insert( Button =>
origin => [15, 150],
    size => [80, 30],
    text => 'Button',
    onClick => sub { $statusbar->text("Button selected") },
);

$mw->insert( ComboBox =>
    origin => [125, 150],
    size => [180, 30],
    name => 'Please, select a color',
    items => [ 'Apricot', 'Beige', 'Black', 'Blue', ],
    style => (cs::DropDown),
    onChange => sub { $statusbar->text("ComboBox selected") },
);

$mw->insert( CheckBox =>
    origin => [15, 95],
    text => 'CheckBox',
    onClick => sub { $statusbar->text("Checkbox selected") },
);

my $inputline = $mw->insert( SpinEdit =>
    origin => [125, 100],
    width => 75,
    min => -459,
    max => 1000,
    editProfile => {color => cl::Black, backColor => cl::White},
    spinProfile => {backColor => cl::LightGray, color => cl::Black},
    onChange => sub { $statusbar->text("SpinEdit selected") },
);

$statusbar = $mw->insert( Label =>
    origin => [0,0],
    text => '',
    size => [ $mw->width, $mw->font->height + 2 ],
    growMode => gm::GrowHiX,
    borderWidth => 1,
    color => cl::Blue,
    backColor => cl::Gray,
);

Prima->run;
```

Listing 17.5: A Status Bar

17.3.3.2 A Status Window

A status window or message window is sometimes more useful for providing feedback. This application features again interactive elements—including a button, combo box, checkbox, and spin edit input—each providing feedback in a dedicated status window, implemented using the `Edit` class, with each entry preceded by the current date and time.

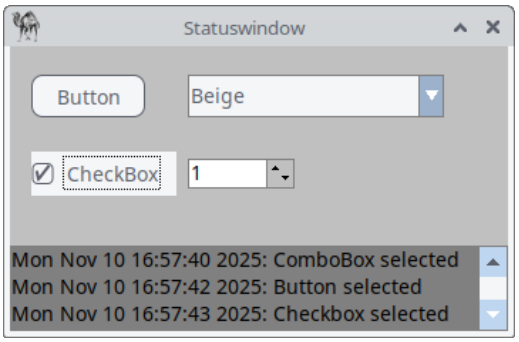


Figure 17.8: A Status Window

```
use Prima qw(Application Label Edit Buttons ComboBox Sliders);

my $statuswindow = '';
my $status_text = '';

sub update_statuswindow {
    my $datestring = localtime();
    $status_text .= "$datestring: $_[0]\n";
    $statuswindow->text($status_text);
}

my $mw = Prima::MainWindow->new(
    text => 'Statuswindow',
    size => [350, 200],
    backColor => cl::LightGray,
    icon => Prima::Icon->load('icon.png'),
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
);

$mw->insert( Button =>
    origin => [15, 150],
    size => [80, 30],
    text => 'Button',
    onClick => sub { update_statuswindow("Button selected"); },
);

$mw->insert( ComboBox =>
    origin => [125, 150],
        size => [180, 30],
        name => 'Please, select a color',
        items => [ 'Apricot', 'Beige', 'Black', 'Blue', ],
        style => (cs::DropDown),
        onChange => sub { update_statuswindow("ComboBox selected"); },
);

$mw->insert( CheckBox =>
    origin => [15, 95],
    text => "CheckBox",
    onClick => sub { update_statuswindow("Checkbox selected"); },
);

my $inputline = $mw->insert( SpinEdit =>
    origin => [125, 100],
    width  => 75,
    min => -459,
    max => 1000,
    editProfile => {color => cl::Black, backColor => cl::White},
    spinProfile => {backColor => cl::LightGray, color => cl::Black},
    onChange => sub { update_statuswindow("SpinEdit selected"); },
);

$statuswindow = $mw->insert( Edit =>
    origin => [0,0],
    size => [ $mw->width, 60],
    text => '',
    color => 0x000000,
    backColor => cl::Gray,
    wordWrap => 1,
    hScroll => 0,
);
```

```
Prima->run;
```

Listing 17.6: A Status Window

17.3.4 When to Use (or Skip) a Panel?

Situation	Use a Panel	Skip it
You want a **framed, decorative section** with multiple child widgets	✅ Yes	
You only need to **display or scroll text**		✅ Yes
You want **border or background control** without extra logic	✅ Yes	
You want **lightweight performance** and no border		✅ Yes

Table 17.2: use a Panel or skip it.

In short, the `Panel` is a *visual tool*, not a requirement. Use it when it clarifies the interface, and leave it out when a simpler widget already provides what you need.

Closing words

In this part you learned how to build the structural backbone of a Prima interface. Containers, the pack manager, and both static and dynamic panes give you precise control over how widgets relate to each other on the screen. GroupBox, Notebook, and Panel add clarity and hierarchy, allowing you to organize content in a way that is both functional and visually intuitive.

Part 7 - Advanced Customization

You've built windows, buttons, and labels - now it's time to supercharge your Prima apps. In Part 7, you'll learn how to:

- Create custom widgets that inherit styles automatically.
- Design interactive tables with sorting, multi-selection, and dynamic updates.
- Load and export data from files, turning your UI into a data powerhouse.

By the end, you'll be crafting apps that look and feel professional - and you'll have the tools to tackle even bigger projects. Ready? Let's dive in!

18. Custom Widgets Through Composition and Inheritance

Want your widgets to automatically match your app's colors and fonts? Or need reusable components that save you time and keep your UI consistent?

In this chapter, you'll learn how to use **inheritance and composition** in Prima to:

- Automatically sync widget styles with `ownerColor` and `ownerBackColor`.
- Create **custom widget classes** with default properties for reusable, uniform designs.
- Mimic inheritance for other properties, like enabling/disabling widgets dynamically.

By the end, you'll be able to build cleaner, more maintainable GUIs without repetitive code.

So: **Build Smarter, Not Harder**

18.1 Understanding `ownerColor` and `ownerBackColor`

Here's an example that demonstrates the use of both `ownerBackColor` and `ownerColor` properties. With these properties, child widgets will automatically inherit the background and foreground colors of the parent widget, allowing for easy color synchronization.

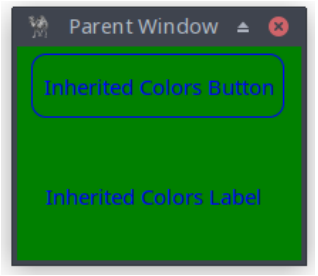


Figure 18.1: Properties ownerColor and ownerBackColor

Click the button 'Inherited Colors Button' and the result is:

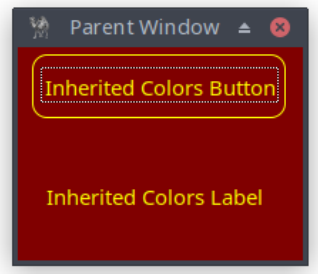


Figure 18.2: Properties ownerColor and ownerBackColor

```
use Prima qw(Application Label Buttons);

# Create the main window, i.e. parent widget
my $mw = Prima::MainWindow->new(
    text      => 'Parent Window',
    size      => [200, 150],
    # Background color for the main window
    color     => cl::LightBlue,
    # Foreground (text) color for the main window
    backColor => cl::Green,
    icon  => Prima::Icon->load('icon.png'),
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
);

# Create a child label that inherits both background and
# foreground colors from the parent
my $child_label = $mw->insert( Label =>
    origin => [20, 30],
    text => 'Inherited Colors Label',
    # Use the parent's background color
    ownerBackColor => 1,
    # Use the parent's text color
    ownerColor     => 1,
);

# Create another child widget (Button) with ownerBackColor and
# ownerColor enabled
my $child_button = $mw->insert( Button =>

    origin => [10, 100],
    text => 'Inherited Colors Button',
    # Use the parent's background color
    ownerBackColor => 1,
    # Use the parent's text color
    ownerColor => 1,

);

# Change parent colors on click to demonstrate automatic color inheritance
$child_button->onClick(
    sub {
            # Change parent's background color
            $mw->color(cl::Yellow);
            # Change parent's foreground (text) color
            $mw->backColor(cl::Red);
    }
);

Prima->run;
```

Listing 18.1: Properties ownerColor and ownerBackColor

The properties *ownerColor* and *ownerBackColor* enable automatic color inheritance. Additionally, the *ownerFont* property allows for automatic font inheritance.

18.2 Creating Reusable Widget Classes with Defaults

ownerColor and *ownerBackColor* are the primary properties specifically designed to enable automatic inheritance of colors from a parent widget.

However, if you want to mimic inheritance for other properties across widgets, there are additional strategies and techniques you can use.

a. Copy properties from the parent when creating the child

You can copy properties from the parent to child widgets when they are created.

```
my $child_button = $mw->insert( Button =>
    text  => 'Copied Properties Button',
    # copy parent's color on initialization
    color => $mw->color,
);
```

This isn't dynamic (changes won't automatically update), but it can be useful for setting initial states.

b. Simulate inheritance with manual propagation

If you want a parent to propagate a property to all children (e.g., *enabled*, size, font…). For example, let's say you want all child widgets to inherit the parent's *enabled* status (whether they are interactive or not). You could create a custom method that applies this property to all child widgets when the parent changes.

```
# Function to propagate 'enabled' status to all children
sub propagate_enabled {
    my ($parent) = @_;
    my $enabled_status = $parent->enabled;
    foreach my $child ($parent->widgets) {
        $child->enabled($enabled_status);
    }
}

# Main window with onClick event to toggle enabled status
$main_window->onClick(sub {
    # Toggle enabled status
    $main_window->enabled(!$main_window->enabled);
    # Apply to all children
    propagate_enabled($main_window);
});
```

c. Best method: define a custom widget class with defaults

However, the best approach is to define a custom class that extends a Prima widget (such as *Prima::Button*) with predefined default properties. Every instance of this class will inherit these defaults, effectively mimicking inheritance.

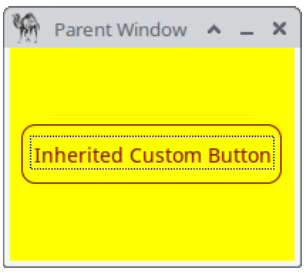


Figure 18.3: Custom Button Class

Define a separate class and save the following code into *my_custom_button.pl*

```
package MyCustomButton;
use Prima::Buttons;
use base 'Prima::Button';

sub profile_default {
    # Get the default profile from the parent class (Prima::Button)
    # and then override only the parts we want to change.
    #
    # %default contains all inherited settings, so we don’t lose
    # important initialization done by Prima::Button. We simply add:
    #   - a custom default text color (color)
    #   - ownerBackColor / ownerColor == 1 so the button uses the
    #     owner's background/foreground when drawing.
    #
    # This ensures the custom widget behaves like a regular button
    # but with our preferred default appearance.
    my %default = %{ $_[ 0 ]->SUPER::profile_default };
    return {
        %default,
        color => cl::LightGreen,
        ownerBackColor => 1,
        ownerColor => 1,

        # Add more defaults if you want:
        # font => { size => 12, style => fs::Bold },
        # centered => 1,
    };
}
# Don’t forget this!
1;
```

Listing 18.2: Custom Button Class

This class acts like 'a normal `Prima::Button`, but with your own defaults.' Invoke this class in your application with `require`:

```
use Prima qw(Application);
use lib '.';
# including the class MyCustomButton
require "my_custom_button.pl";

my $mw = Prima::MainWindow->new(
    text => 'Parent Window',
    size => [200, 150],
    color => cl::Red,
    backColor => cl::Yellow,
    icon => Prima::Icon->load('icon.png'),
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
);

my $custom_button = $mw->insert('MyCustomButton' =>
    centered => 1,
    text => 'Inherited Custom Button',
);

Prima->run;
```

Listing 18.3: Invoke Custom Class With `require`

If we set in `my_custom_button.pl`

```
ownerBackColor => 0,
ownerColor => 0,
```

then the result is as expected: LightGreen text color and default system default background.

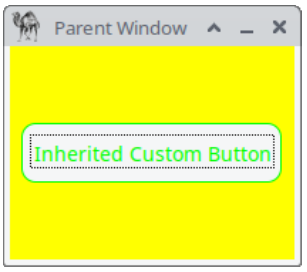


Figure 18.4: Got No Colors from Parent

To summarize, a custom class gives you:

- Reusable default properties
- Inheritable color behavior
- Uniform style across your app
- Cleaner code in your main program

19. Working with Table Widgets

19.1 Grid Widget Overview

`Prima::Grids` is a versatile widget for displaying and managing tabular data in Perl Prima applications. Building on `Prima::GridViewer` and `Prima::AbstractGridViewer`, it provides functionality to add, delete, and insert rows and columns, with customizable cell styles, adjustable row/column dimensions, and colored gridlines. It also supports data selection and processing, making it ideal for dynamic and interactive data presentation.

19.2 Basic example

This Perl program creates a simple GUI window, which displays a 2×3 grid with fixed-size cells, where each cell contains a value, and the first cell in the first row is focused by default. The default font is applied to the grid.

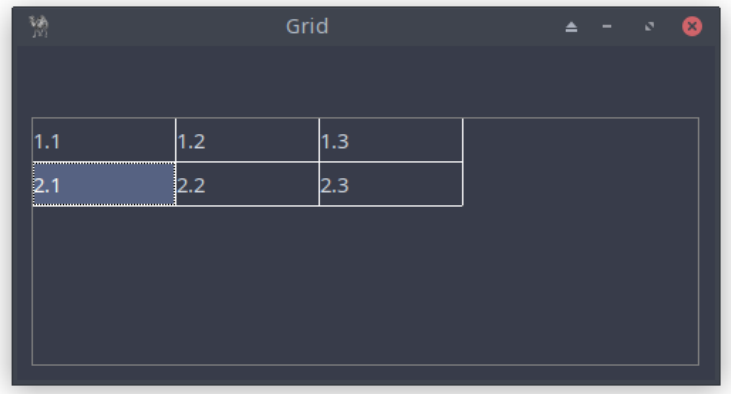


Figure 19.1: Grid

```
use Prima qw(Grids Label Application);

my $mw = Prima::MainWindow->new(
    text => 'Grid',
    size => [320, 250],
    font => { size => 11, },
    icon  => Prima::Icon->load('icon.png'),
);

my $grid = $mw->insert ( Grid =>

    origin => [10, 10],
    size => [470,175],
    ownerFont => 1,                  # Grid uses the window font
    columns => 3,                    # Number of columns in the grid
    constantCellHeight => 30,        # All rows have fixed height = 30
    constantCellWidth => 100,        # All columns have fixed width = 100
    allowChangeCellHeight => 0,      # Disallow resizing of rows
    allowChangeCellWidth => 0,       # Disallow resizing of columns
    cells => [
                [qw(1.1 1.2 1.3)],   # Row 0 with 3 cells
                [qw(2.1 2.2 2.3)],   # Row 1 with 3 cells
            ],
    gridColor => cl::White,          # Color grid lines (default cl::Black)
    focusedCell => [0,1],            # Cell initially focused: row 0, column 1 (second cell)
);

Prima->run;
```

Listing 19.1: Grid

19.3 Adding & Removing Rows and Columns

Adding or deleting a row or column can be easily achieved using four built-in functions.

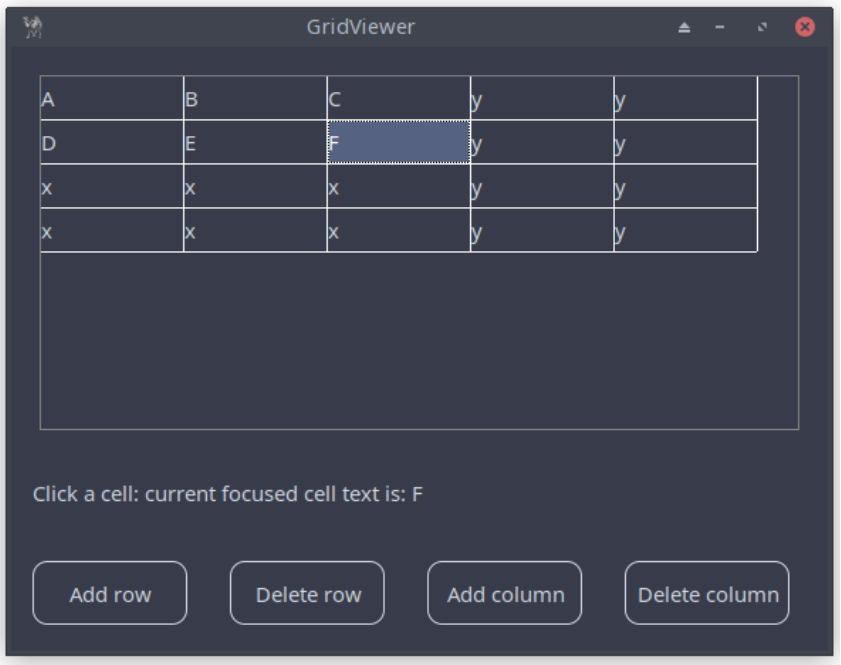


Figure 19.2: Grid With Extra Functionality

Function	Description	Example
`add_column @ITEMS`	Accepts an array of strings (or list) representing a single column. Appends the column to the end of the existing columns.	`add_column('Row 1 Col 3', 'Row 2 Col 3', 'Row 3 Col 3');`
`add_row @ITEMS`	Accepts an array of strings (or list) representing a single row. Appends the row to the end of the existing rows.	`add_row('Row 2 Col 1', 'Row 2 Col 2', 'Row 2 Col 3');`
`delete_columns OFFSET, COUNT`	Removes `COUNT` columns starting from `OFFSET` (index of the first column to remove). Negative `OFFSET` values are accepted.	`# Removes the second column delete_columns(1, 1);`
`delete_rows OFFSET, COUNT`	Removes `COUNT` rows starting from `OFFSET` (index of the first row to remove). Negative `OFFSET` values are accepted.	`# Removes the second row delete_rows(1, 1);`

Table 19.1: 4 Useful Grid Functions

```
use Prima qw(Grids Label Buttons Application);

my $mw = Prima::MainWindow->new(
    text => 'GridViewer',
    size => [575, 425],
    font => { size => 11, },
    icon  => Prima::Icon->load('icon.png'),
);

my $label;

my $grid = $mw->insert ( Grid =>
    pack => { fill => 'x', side => 'top', pad => 40 },
    size => [525,250],
    ownerFont => 1,
    columns => 3,
    cells => [
                [qw(A B C)],
                [qw(D E F)],
             ],

    constantCellHeight => 30,
    constantCellWidth => 100,
    allowChangeCellHeight => 0,
    allowChangeCellWidth => 0,

    # default cl::Black
    gridColor => cl::White,
    # row 0, column 0, i.e. [0, 0] is focused by default, now [0, 1]
    focusedCell => [0, 1],

    onSelectCell => sub {
        $label->text("Click a cell: current focused cell text is: " .
                            $_[0]->get_cell_text($_[0]->focusedCell) );

    },

);

$label = $mw->insert( Label =>
    pack => { fill => 'x', side => 'top', pad => 30 },
    text => "Click a cell: current focused cell text is: " .
                              $grid->get_cell_text($grid->focusedCell) ,
);

# four buttons with some onClick actions
# default geometry settings for the four buttons

my %mcPack = (pack => { fill => 'x', side => 'left', pad => 30 } );

$mw->insert( Button =>
    %mcPack,
    text => "Add row",
    onClick => sub {
                         my @array = ('x') x $grid->columns;
                         $grid->add_row(@array);
                           },
);

$mw->insert( Button =>
    %mcPack,
```

```
    text => "Delete row",
    hint => "Delete last row",
    onClick => sub {
                # delete last row
                $grid->delete_rows($grid->rows-1,1) if ($grid->rows > 0);
    },

);

$mw->insert( Button =>
    %mcPack,
    text => "Add column",
    onClick => sub {
                        my @array = ('y') x $grid->rows;
                        $grid->add_column(@array);
    },

);

$mw->insert( Button =>
    %mcPack,
    text => "Delete column",
    hint => "Delete last column",
    onClick => sub {
         # delete last column
         $grid->delete_columns($grid->columns-1,1) if ($grid->columns > 0);
    },
);

Prima->run;
```

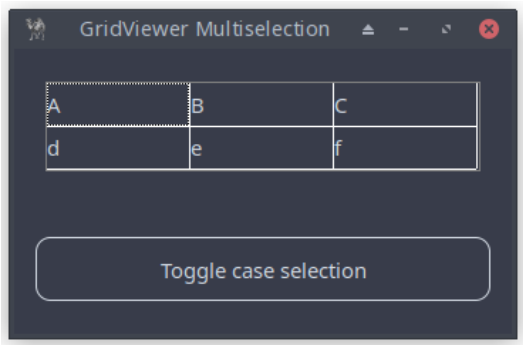
Listing 19.2 : Grid With Extra Functionality

The other methods on adding and removing rows and columns are well-documented - give them a try! Next section on multiselection and processing the cell data.

19.4 Multi-Selection and Dynamic Updates

This small application shows a grid where you can select multiple cells and toggle their case with a button. The window displays a 3×2 grid filled with letters. Simply click and drag to select one or more cells, then press “Toggle case selection” to switch lowercase letters to uppercase and vice versa.

The script features a main window containing a grid widget that supports multiselection (*`multiSelect => 1`*), allowing users to toggle the case of text within selected cells. The script defines a helper function to process rectangular cell selections efficiently *`get_cells_in_rectangle(\@selection)`*.



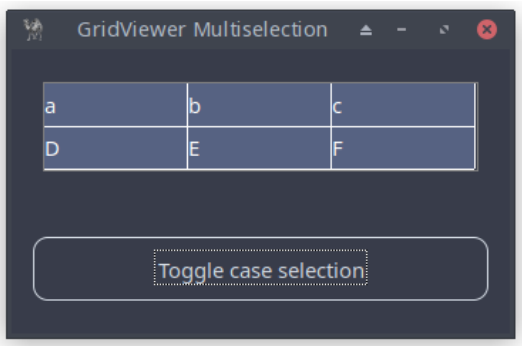


Figure 19.3: Grid With Extra Functionality

```
use Prima qw(Application Buttons Grids);

my $mw = Prima::MainWindow->new(
    text => 'GridViewer Multiselection',
    size => [350, 200],
    font => { size => 11, },
    icon  => Prima::Icon->load('icon.png'),
);

my $grid = $mw->insert( Grid =>
    pack => { expand => 1, fill => 'none', pad => 30 },
    size => [306,63],
    columns => 3,
    cells => [
                [qw(A B C)],
                [qw(d e f)],
             ],

    constantCellHeight => 30,
    constantCellWidth => 100,
    allowChangeCellHeight => 0,
    allowChangeCellWidth => 0,

    multiSelect => 1,

    gridColor => cl::White,
);

$mw->insert( Button =>

    pack => { expand => 1, fill => 'x', side => 'bottom', pad => 30 } ,
    text => "Toggle case selection",
    onClick => sub {
                # Retrieve the selection bounds: top-left (x1, y1) and
            # bottom-right (x2, y2)
                my ($x1, $y1, $x2, $y2) = $grid->selection;
                return print "No cells selected\n" if $x1 < 0;
                for ( my $x = $x1; $x <= $x2; $x++) {
                                for ( my $y = $y1; $y <= $y2; $y++) {
                            # Retrieve the current value of the cell
                            # at position (x, y).
                            my $cell = $grid->cell($x, $y);
                            $grid->cell($x, $y, ($cell =~ /^[a-z]$/) ?
                                                        uc($cell) :
                                                        lc($cell) );
                 }
             }
             # Refresh the grid to display the updated cell values
             $grid->repaint;
     }
);

Prima->run;
```

Listing 19.3: Grid With Extra Functionality

20. DetailedList Widgets

20.1 Understanding `DetailedList`

`Prima::DetailedList` is a widget that provides a flexible and feature-rich table view for displaying tabular data. It resembles a spreadsheet or table and is commonly used for showing multi-column lists with sorting, selection, and scrolling capabilities. Key features include:

- Multiple columns: each entry can have multiple fields, displayed across columns.
- Customizable cells: each cell can contain different types of data and support custom rendering.
- Sorting: allows sorting of rows.
- Selectable rows: users can select one or more rows

This widget is often used in applications where users need to view, sort, and interact with structured data in a visually organized way.

A basic example:

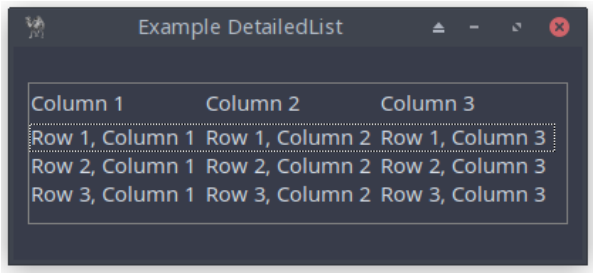


Figure 20.1: DetailedList

20.2 Building Your First DetailedList Interface

How to build the DetailedList application introduced previously and how to cutomize it? Let’s go step by step in the following sections, asking questions and finding answers as we progress.

We'll need properties for:

- displaying items in columns
- showing column headers

along with a first instruction on how to use the application.

Well, the https://metacpan.org/pod/Prima::DetailedList is fully clear, how to do that:

Property	Description
items (ARRAY)	An array of scalar arrays (default: strings). Data flows left to right, top to bottom.
columns (INTEGER)	Defines the number of columns in items. If changed, both items and headers are restructured. Default: 0
headers (ARRAY)	An array of strings for column titles in the header widget.

Table 20.1: Some Properties DetailedList

The instruction is placed in a Label class, which displays a static message explaining that you can click on the column header to sort.

Figure 20.2: DetailedList

```
use Prima qw(DetailedList Label Application);

my $mw = Prima::MainWindow->new(
    text => 'DetailedList',
    size => [500, 250],
    font => { size => 11, },
    icon  => Prima::Icon->load('icon.png'),
);

$mw->insert( Label =>
    pack => { side => 'top', fill => 'x', pady => 30 },
    # instruction text
    text => "  Click the column header to sort!",
    alignment  => ta::Left,
    ownerFont => 1,
);

my $d_list = $mw->insert( 'Prima::DetailedList' =>

        pack => { side => 'left', fill => 'none', pad => 15 },
        size => [480,175],

        # manages the number of columns in items
        columns => 3,

        # an array of strings passed to the header widget as column titles
        headers => [ 'Name', 'Year of birth', 'Year of death' ],

        # an array of arrays of scalars of any kind (string are assumed)
        items => [
            ['Bach, Johann Sebastian', 1685, 1750],
            ['Bruckner, Anton', 1824, 1896],
            ['Beethoven, Ludwig von', 1770, 1827],
            ['Borodin, Alexander', 1833, 1887],
            ['Berg, Alban', 1885, 1935],
            ['Berlioz, Hector', 1803, 1869],
            ['Bizet Georges', 1838, 1875],
        ],

);

Prima->run;
```

Listing 20.1: DetailedList

20.3 Customizing Columns and Headers

Upon reviewing the previous example, one might have suggestions to gave it a better look for header and items. Well, reading https://metacpan.org/pod/Prima::DetailedList gaves you information on its customizing (via headerClass and headerProfile – see 20.5) but also implicitly important additional information:

- Prima::DetailedList is a descendant of Prima::ListViewer and as such also provides a certain level of abstraction. It overloads the format of items in order to support multi-column (2D) cell span.
- Prima::DetailedList also inserts Prima::Widget::Header widget on top of the list so that the user can interactively move, resize, and sort the content of the list. In other words, it is a controlling header widget

This means that a lot of properties and methods are inherited. Let's have a look at Prima::ListViewer . As we look at https://metacpan.org/release/KARASIK/Prima-1.74/source/Prima/Lists.pm , it inherits from its parent Prima::AbstractListViewer which in turn inherits from

- Prima::Widget
- Prima::Widget::Fader
- Prima::Widget::GroupScroller
- Prima::Widget::ListBoxUtils
- Prima::Widget::MouseScroller

`Prima::Widget::Header` also inherits from `Prima::Widget`. For now, the `Prima::Widget` class is the most important to examine for general inherited properties and methods. We will not repeat here (see …) but will focus on two key activities: customizing the header and the items.

20.3.1 Customizing the Header

When we look at Section 20.4 and 20.5 on customizing the header, we now know where to focus:

1. Prima::DetailedList

Property	Description	Default Value
headerClass	Assigns the header class.	`Prima::Widget::Header`
headerProfile	Assigns a hash of properties passed to the header widget during creation.	N/A (depends on values)

Table 20.2: Header Properties of Class DetailedList

Note: `headerClass` is an advanced topic. It is mentioned briefly but will be skipped for now.

2. Prima::Widget::Header

Property	Type	Description	Default Value
clickable	BOOLEAN	Selects if the user is allowed to click the tabs.	1
draggable	BOOLEAN	Selects if the user is allowed to move the tabs.	1
pressed	INTEGER	Contains the index of the currently pressed tab. `-1` when no tabs are pressed.	-1
scalable	BOOLEAN	Selects if the user is allowed to resize the	1

Property	Type	Description	Default Value
		tabs.	
widths	ARRAY	Array of integer values corresponding to the extents of the tabs (widths or heights depending on `vertical`).	N/A

Table 20.3: Header Properties and Methods of Class Widget::Header

Excluded for readability: `items`, `minTabWidth`, `offset`, `vertical`.

20.3.2 Customizing the Items

When we look at Section 20.6 on customizing the items, we also know where to focus:

1. Prima::AbstractListViewer

Property	Type	Description	Default Value
autoHeight	BOOLEAN	If 1, adjusts item height automatically when the widget's font changes. Useful for text items.	1
count	INTEGER	The number of items in the list. Often read-only in descendants.	-
draggable	BOOLEAN	If 1, allows items to be dragged interactively using Ctrl + left mouse button.	0
drawGrid	BOOLEAN	If 1, vertical grid lines between columns are drawn.	0
extendedSelect	BOOLEAN	Controls multi-selection behavior when `multiSelect` = 1. Enables drag selection, Shift+arrows, Ctrl for toggling.	0
focusedItem	INDEX	Index of the focused item. `-1`	-1

Property	Type	Description	Default Value
		if none.	
gridColor	COLOR	Color used for vertical divider lines in multi-column lists.	`cl::Black`
multiSelect	BOOLEAN	If 0, only one item can be selected. If 1, multiple items may be selected.	0
selectedCount	INTEGER	Read-only; returns the number of selected items.	-
selectedItems	ARRAY	Array of integer indices for selected items.	-

Table 20.4: Properties of Class AbstractListViewer

Excluded for readability: `integralHeight`, `integralWidth`, `itemHeight`, `itemWidth`, `multiColumn`, `offset`, `topItem`, `vertical`.

2. Prima::ListViewer

Property	Type	Description	Default Value
autoWidth	BOOLEAN	Recalculates item width automatically when the font or item list changes.	1
count	INTEGER	Read-only; returns number of items.	N/A
items	ARRAY	Accesses the storage array of items. Format not defined—treated as one scalar per index.	N/A

Table 20.5: Properties of Class ListViewer

20.4 Sorting, Alignment, and Spacing

Upon reviewing the previous example in 20.2, one might have the following suggestions:

- the column headers could have more space
- sorting should be applied to the first column.

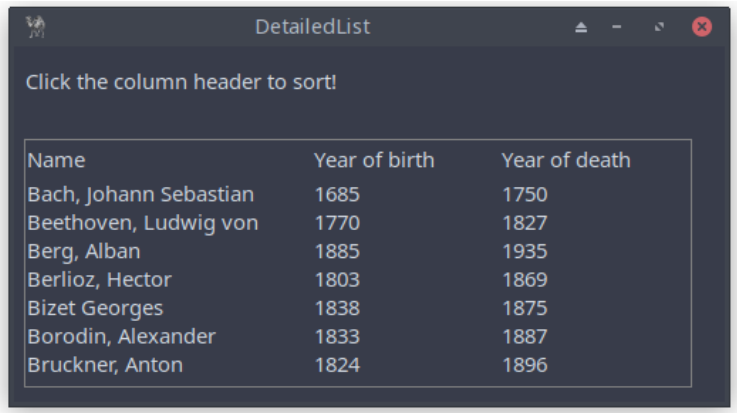


Figure 20.3: DetailedList Class with Header Modifications, Left-Aligned, and Sorted Items

```
use Prima qw(DetailedList Label Application);

my $mw = Prima::MainWindow->new(
    text => 'DetailedList',
    size => [500, 250],
    font => { size => 11,  },
    icon  => Prima::Icon->load('icon.png'),
);

$mw->insert( Label =>
    pack => { side => 'top', fill => 'x', pady => 30 },
    text       => "  Click the column header to sort!",
    alignment  => ta::Left,
    ownerFont => 1,
);

my $d_list = $mw->insert( 'Prima::DetailedList' =>

        pack => { side => 'left', fill => 'none', pad => 15 },
        size => [480,175],
        # manages the number of columns in items
        columns => 3,

        # from: Prima::Widget::Header
        # widths ARRAY
        # Array of integer values, corresponding to the extents of the tabs.
        widths => [200, 130, 130],

        # an array of strings passed to the header widget as column titles
        headers => [ 'Name', 'Year of birth', 'Year of death' ],

        # an array of arrays of scalars of any kind (string are assumed)
        items => [
            ['Bach, Johann Sebastian', 1685, 1750],
            ['Bruckner, Anton', 1824, 1896],
            ['Beethoven, Ludwig von', 1770, 1827],
            ['Borodin, Alexander', 1833, 1887],
            ['Berg, Alban', 1885, 1935],
            ['Berlioz, Hector', 1803, 1869],
            ['Bizet Georges', 1838, 1875],
        ],

       # aligns is not necessary, but here for demonstration purposes only.
        # an array of the ta:: align constants where each defines the column
        # alignment.
        # ta::Left, ta::Right, ta::Center

        aligns => [ta::Left, ta::Left, ta::Left,],

);

# sorts items by the column index in ascending order, in this case, "Name" (index 0).
$d_list->sort(0);

Prima->run;
```

Listing 20.2: DetailedList Class with Header Modifications, Left-Aligned, and Sorted Items

20.5 Customizing Header

Another suggestion is to make the header have a different background color and bold text.

The `headerProfile` property of **Prima::DetailedList** is a hash used to configure various settings for the header widget (a `Prima::Widget::Header` widget) that displays the column headers. By customizing `headerProfile`, you can set properties like font, color, alignment, and more—independently of the main DetailedList.

Below is a table summarizing commonly used **Widget properties** that can be set within the `headerProfile` of `Prima::DetailedList`:

Property	Description	Example Values
font	Customize font properties (name, size, style) for header text.	`{ size => 14, style => fs::Bold }`
backColor	Set the background color of the header.	`cl::White`, `cl::Gray`, `cl::Red`
color	Set the text color of the header.	`cl::Black`, `cl::Red`, `cl::Blue`, `cl::Green`
alignment	Control the text alignment within header cells.	`ta::Left`, `ta::Center`, `ta::Right`

Table 20.6: Header Profile Properties

Applying these properties…

```
headerProfile => { backColor => cl::LightGray,
                   color => 0x00003f,
                   font => { name => 'Courier New', size => 11, },
                   alignment => ta::Left
},
```

to source code of figure 20.4, results in the following:

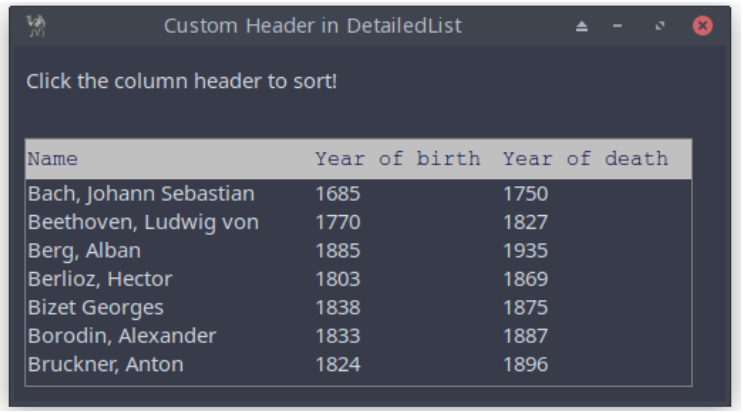


Figure 20.4: Basic Application of the DetailedList Class with Header Modifications

20.6 Customizing Items

We can add some properties from `Prima::AbstractListviewer`:

```
gridColor => cl::Red,
drawGrid => 1,
```

and vertical grid lines between columns are drawn. In that case, you could implement:

```
aligns => [ta::Left, ta::Center, ta::Center,],
```

As already said, *Prima::DetailedList* inherits properties and methods of these Widget classes.

A few examples of properties and methods from the class Widget, which have effect on the items of DetailedList (not the header: see next section):

```
# background color
backColor => cl::LightGray,

# text color
color => cl::Blue,

# if 1, the first item gets the focus
current => 1,

# make items not clickable
enabled => 0,

# add a hint message
hint => "All are brilliant composers, but Bach is the best!",

# click an item
onMouseClick => sub {
          my @item = @{$_[0]->items->[$_[0]->focusedItem]};
          my $str = $item[0] . " (" . $item[1] . " - " . $item[2] . ")";
      showMessageDialog(400, 125, "You clicked...", "You clicked:\n\n$str")
},
```

You'll see the following when *backColor*, *color*, *hint* and *onMouseClick* are implemented:

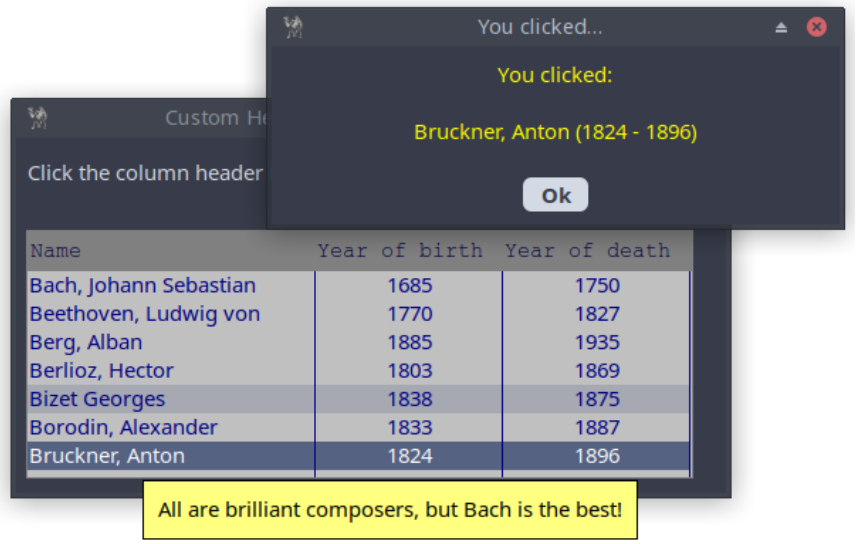


Figure 20.5: DetailedList application With Additional Features

Alternatively, you could implement:

```
onMouseClick => sub {
    my ($self, $index, $state) = @_;
    $index = $index->[0] if (ref $index eq 'ARRAY');
    # Ensure index is valid and proceed
    if (defined $index && $state) {

        my $str = @{$self->items->[$index]}[0] . " (" .
                  @{$self->items->[$index]}[1] . " - " .
                  @{$self->items->[$index]}[2] . ")";
        showMessageDialog(400, 125, "You clicked...",
                                                "You clicked:\n\n$str");
    }
},
```

20.7 headerClass property

You can use the property *`headerClass`* to customize the header's behavior, appearance, or properties if you want to extend or replace the default header.

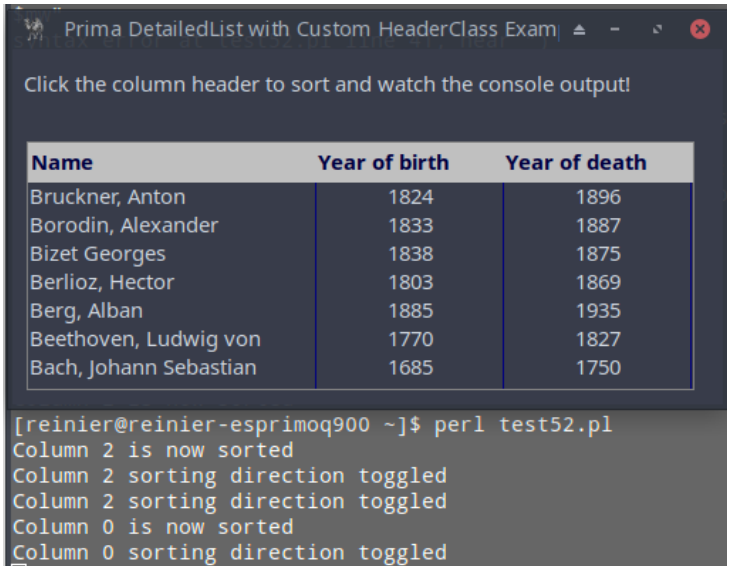


Figure 20.6: DetailedList Application with Custom Header

```
use Prima qw(Application Label DetailedList);

# Define a custom header class
package MyCustomHeader;
# ... that inherits from Prima::Header
use base 'Prima::Widget::Header';

# Add a property to keep track of the last sorted column
sub init {
    my $self = shift;
    $self->{last_sorted_column} = -1;
    $self->SUPER::init(@_);
}

# Override on_click to change column behavior on click
sub on_click {
    my ($self, $column_index) = @_;

    # Toggle sorting indication
    if ($self->{last_sorted_column} == $column_index) {
        print "Column $column_index sorting direction toggled\n";
    } else {
        print "Column $column_index is now sorted\n";
    }

    $self->{last_sorted_column} = $column_index;

    # Continue with the default on_click behavior
    $self->SUPER::on_click($column_index);
}

package Main;

my $mw = Prima::MainWindow->new(
    text => 'Prima DetailedList with Custom HeaderClass Example',
    size => [500, 250],
    font => { size => 11, },
    icon  => Prima::Icon->load('icon.png'),
);

$mw->insert( Label =>
    pack => { side => 'top', fill => 'x', pady => 30 },
    text => "  Click the column header to sort and watch the " .
                                                    "console output!",
    alignment  => ta::Left,
);

$mw->insert( DetailedList =>

      origin => [10, 10],
      size => [470,175],

      # Use MyCustomHeader as the headerClass
      headerClass  => 'MyCustomHeader',
      headerProfile => {
                              backColor => cl::LightGray,
                              color => 0x00003f,
                              font => { size => 11, style => fs::Bold  },
                              alignment => ta::Left,
                           },
```

```
      columns => 3,
      widths      => [200, 130, 130],
      headers     => ['Name', 'Year of birth', 'Year of death'],

      items => [
            ['Bach, Johann Sebastian', 1685, 1750],
            ['Bruckner, Anton', 1824, 1896],
            ['Beethoven, Ludwig von', 1770, 1827],
            ['Borodin, Alexander', 1833, 1887],
            ['Berg, Alban', 1885, 1935],
            ['Berlioz, Hector', 1803, 1869],
            ['Bizet Georges', 1838, 1875],
      ],

      aligns => [ta::Left, ta::Center, ta::Center,],
      gridColor => cl::Blue,
      drawGrid => 1,
);

Prima->run;
```

Listing 20.3: DetailedList With Custom Header

20.8 Populating and Rearranging Items from a File

This script reads composer data from a CSV file and displays it in a GUI table using `Prima::DetailedList`. It begins by parsing the CSV to separate the header from the data, then creates a window with a sortable table that shows each composer's name, birth year, and death year. You can view, sort, and interact with the data. Additionally, with the `draggable` setting enabled, you can hold CTRL and drag items with the left mouse button to rearrange rows, making it easy to visually reorganize the data.

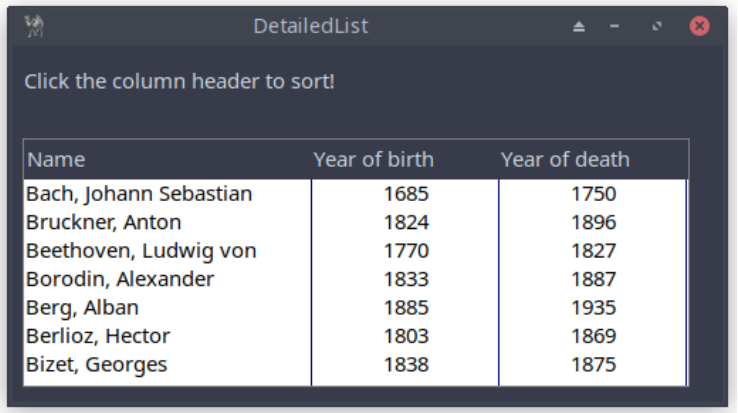


Figure 20.7: DetailedList Application Using a CSV-file

```
use lib '.';
use Text::CSV;

#-----------------------------------------------------#
# CSV file: header on line 1 and data on the next lines
# ; to separate fields

=begin
Name;Year of birth;Year of death
Bach, Johann Sebastian;1685;1750
Bruckner, Anton;1824;1896
Beethoven, Ludwig von;1770;1827
Borodin, Alexander;1833;1887
Berg, Alban;1885;1935
Berlioz, Hector;1803;1869
Bizet, Georges;1838;1875
=cut

# Open CSV file
my $file = 'composers.csv';
open my $fh, '<', $file or die "Could not open '$file' $!\n";

# Create a CSV parser
# the ; used to separate fields
my $csv = Text::CSV->new({ sep_char => ";",  binary => 1, auto_diag => 1 });

# Arrays to hold the data
my @header;
my @data;

# counter variable
my $row_count = 0;

# Parse each line of the CSV file
while (my $row = $csv->getline($fh)) {

        if ($row_count == 0) {
            push(@header, $row);
        $row_count++;
        } else {
        push(@data, $row);
    }
}

# Close the file handle
close $fh;

#-----------------------------------------------------#

use Prima qw(DetailedList Label Application);

my $mw = Prima::MainWindow->new(
    text => 'DetailedList',
    size => [500, 250],
    font => { size => 11, },
    icon  => Prima::Icon->load('icon.png'),
);

$mw->insert( Label =>
    pack => { side => 'top', fill => 'x', pady => 30 },
    text => "  Click the column header to sort!",
```

```
    alignment  => ta::Left,
    ownerFont => 1,
);

my $d_list = $mw->insert( 'Prima::DetailedList' =>

        pack => { side => 'left', fill => 'none', pad => 15 },

        size => [470,175],

        columns => 3,
        # @header is array of strings
        headers => @header,
        widths => [200, 130, 130],
        # reference to @data
        items => \@data,

        backColor => 0xFFFFFF,
        color => 0x000000,

        gridColor => cl::Blue,
        drawGrid => 1,
        aligns => [ta::Left, ta::Center, ta::Center,],

        # move items via CTRL + Left mouse
        draggable => 1,

);

Prima->run;
```

Listing 20.4: DetailedList Application Using a CSV-file

Add some functionality to export the list of items.

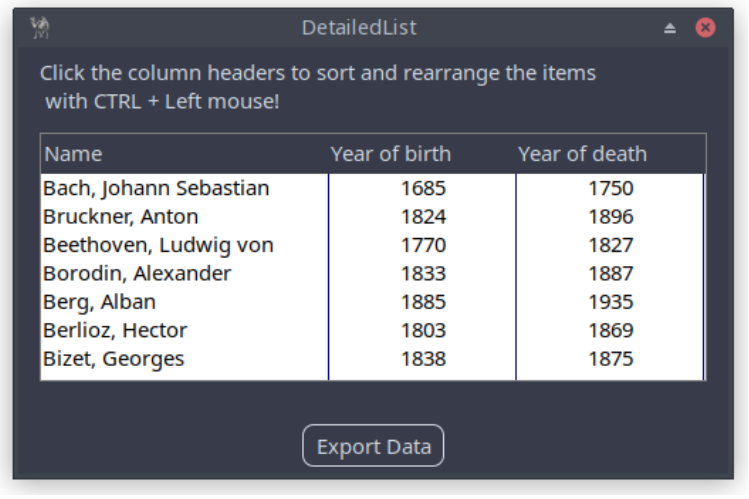


Figure 20.8: DetailedList Application With Button to Export Rearranged Data

```
# Step 1: Add a button to export rearranged data
$mw->insert( Button =>
    origin => [500/2-50, 5],
    size => [100, 30],
    text => 'Export Data',
    onClick => sub {
        export_rearranged_data($d_list, 'rearranged_composers.csv');
    },
);

# Function to export data to a CSV file
sub export_rearranged_data {
    my ($list_widget, $filename) = @_;

    # Create CSV object
    my $csv = Text::CSV->new({ sep_char => ";", binary => 1 });

    # Open the file to write
    open my $fh, '>', $filename or die "Could not open '$filename' $!\n";

    # Write header
    $csv->print($fh, $header[0]);
    print $fh "\n";

    # Write each item in the rearranged order
    for my $item (@{$list_widget->items}) {
        $csv->print($fh, $item);
        print $fh "\n";
    }

    close $fh;
    print "Data exported successfully to $filename\n";
}
```

20.9 More on sorting

The data can be sorted by clicking the column header. In some cases, it may be necessary to customize the sorting algorithm. Here's how it can be done.

```
my $arr = [
    ['Philipp','30.12.1777','18.11.1840'],
    ['Catharina','1672','13.10.1745'],
    ['Georg','05.07.1843','-'],
    ['Christina','-','11.01.1785'],
    ['Andreas','08.03.1801','18.01.1835'],
];
```

We use the following data. Some of them are missing, some are incomplete.

```
my $arr = [
    ['Philipp','30.12.1777','18.11.1840'],
    ['Catharina','99.99.1672','13.10.1745'],
    ['Georg','05.07.1843','99.99.9999'],
    ['Christina','99.99.9999','11.01.1785'],
    ['Andreas','08.03.1801','18.01.1835'],
];
```

We want to sort on year. So, we extract the year from the data (e.g. from the second column), and use the spaceship operator (<=>) for custom numerica sorting. For alphabetic comparison, the cmp operator can be used.

The year can be extracted using a regex operation, as shown below:

```
$second_column_date = '30.12.1777';
my ($a_year) = $second_column =~ /(\d{4})$/;
```

To perform a numeric comparison, consider the following example:

```
my @years = (2020, 2015, 2022, 2018);
# Sorts numerically in ascending order
my @sorted_years = sort { $a <=> $b } @years;
```

The next thing we do is adding a *onSort* event *directly in the widget definition.* The complete code:

```
use Prima qw(Application DetailedList);
my $mw = Prima::MainWindow->new(
    text => 'DetailedList Sorting Example',
    size => [400, 300],
);
my $arr = [
    ['Philipp','30.12.1777','18.11.1840'],
    ['Catharina','99.99.1672','13.10.1745'],
    ['Georg','05.07.1843','99.99.9999'],
    ['Christina','99.99.9999','11.01.1785'],
    ['Andreas','08.03.1801','18.01.1835'],
];

$mw->insert( 'DetailedList' =>
    pack => { expand => 1, fill => 'both' },
    columns => 3,
    headers => [ 'Name', 'Birth date', 'Death date' ],
    items => $arr,
    onSort => sub {
        my ($self, $column_index, $order) = @_;
        # If the callback procedure is willing to sort by COLUMN index, then it must
        # call clear_event to signal that the event flow must stop.
        $self->clear_event();
        # Debugging output
        print "Sorting column $column_index in order $order\n";
        my @sorted_items;
        if ($column_index == 0) {
            @sorted_items = sort {
                  $a->[0] cmp $b->[0]
            } @{$self->{items}};
        } elsif ($column_index == 1) {
            @sorted_items = sort {
                # Extract year
                my ($a_year) = $a->[1] =~ /(\d{4})$/;
                my ($b_year) = $b->[1] =~ /(\d{4})$/;
                # Compare by the year in the second column
                $a_year <=> $b_year;
            } @{$self->{items}};
        } elsif ($column_index == 2) {
             @sorted_items = sort {
                 # Extract year
                 my ($a_year) = $a->[2] =~ /(\d{4})$/;
                 my ($b_year) = $b->[2] =~ /(\d{4})$/;
                 # Compare by the year in the third column
                 $a_year <=> $b_year;
             } @{$self->{items}};
         }
         # Reverse the order if the user clicked to sort in descending order
         @sorted_items = reverse @sorted_items if ($order == 1);
         # Update the items in the DetailedList with the sorted data
         $self->items(\@sorted_items);
    },
);
Prima->run;
```

Listing 20.5: DetailedList Application With Button to Export Rearranged Data

Closing words

Advanced customization in Prima empowers you to:

- **Create cohesive UIs** with consistent styling and behavior.
- **Build dynamic applications** that respond to user input and data changes.
- **Extend functionality** by defining custom widget classes and event handlers.

Prima's flexibility and extensibility make it a powerful tool for building desktop applications in Perl. By mastering these advanced customization techniques, you can create applications that are not only functional but also visually appealing and user-friendly.

Part 8 - Extending Prima with Your Own Logic

21. Strategies for Extending Prima Applications

Prima is a flexible toolkit that's easy to customize. Whether you're working with graphics, windows, or system tools, Prima gives you control over how things behave. You're not locked into using everything exactly as it comes—you can build on top of it or change parts to fit your needs.

Most of this customization happens through Prima's object-oriented interface in Perl. Perl makes it easy to add, change, or even replace methods - sometimes while your program is running. This means you can shape Prima around your application, instead of shaping your application around the toolkit.

In addition to writing your own code, you can also extend Prima using external Perl modules from CPAN. Many CPAN libraries work well alongside Prima—for example, for tasks like networking, file handling, or advanced data structures. While this chapter focuses on ways to customize Prima itself, don't forget that CPAN is a powerful tool for adding features to your application (I'll show you one example below).

Let's have a look at a few simple ways to extend and modify how Prima works:

1. **Using Event Callbacks**
   This is the easiest way to customize behavior. You can respond to things like mouse clicks, key presses, or repainting a window by using event handlers such as `onClick`, `onKeyDown`, or `onPaint`.
2. **Subclassing and Overriding Methods**
   A more powerful technique where you create your own versions of Prima's classes. This lets you add new features or change how something works by overriding its built-in methods.
3. **Using Composition (Has-a Relationship)**
   Instead of creating a new class from an existing one, you can make your own objects that *contain* Prima widgets. This helps you build more flexible or reusable parts without needing inheritance.
4. **Mixin Modules (Sharing Code Between Classes)**
   If you have behavior that's useful in several places, you can move it into a separate module and include it wherever it's needed. This avoids repeating code and keeps things organized.

Before we dive into these techniques, we'll take a quick look at some key ideas from object-oriented programming (OOP). Understanding these will make it easier to work with Prima and get the most out of its features.

21.1 Extension strategy 1: Using Event Callbacks

This example demonstrates modular event handling in Prima by:

1. Defining reusable subroutines (`handle_button_click`, `handle_window_paint`) for button clicks and window painting.
2. Assigning these subroutines as event handlers (`onClick`, `onPaint`) to a button and the main window.
3. Updating the button's text to "Clicked!" and printing a message to the console when clicked.

Purpose: Show how to separate event logic from widget creation for cleaner, reusable, and maintainable code.

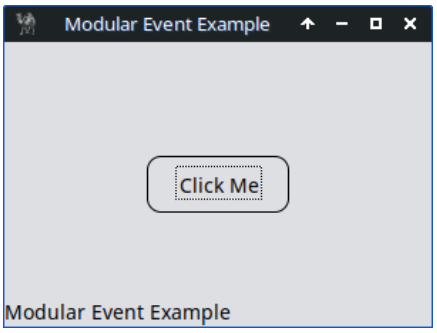


Figure 21.1: Modular Event Handling, Button Click Example

```
use Prima qw(Application Buttons);

# Modular event handler for button clicks
sub handle_button_click {
    my ($button) = @_;
    $button->text("Clicked!");
    print "Button was clicked!\n";
}

# Modular event handler for window paint
sub handle_window_paint {
    my ($window, $canvas) = @_;
    $canvas->clear;
    $canvas->text_shape_out($window->text, 0, 0);
}

# Main application
my $mw = Prima::MainWindow->new(
    text    => 'Modular Event Example',
    size    => [300, 200],
    icon   => Prima::Icon->load('icon.png'),
    onPaint => \&handle_window_paint,
);

$mw->insert(
    Button =>
        text    => 'Click Me',
        origin  => [100, 80],
        size    => [100, 40],
        onClick => sub { handle_button_click(@_); },
);

Prima->run;
```

Listing 21.1: Modular Event Handling, Button Click Example

This next example demonstrates dynamic event handling in Prima by:

1. Storing event handlers (e.g., `on_click`, `on_paint`) in a hash (`%event_handlers`).
2. Assigning these handlers to widgets (e.g., a button and window).
3. Changing the button's text from "Click Me" to "I Was Clicked!" when clicked, providing visual feedback that the event worked.

Purpose: Show how to centralize, reuse, and dynamically update event logic in Prima applications.

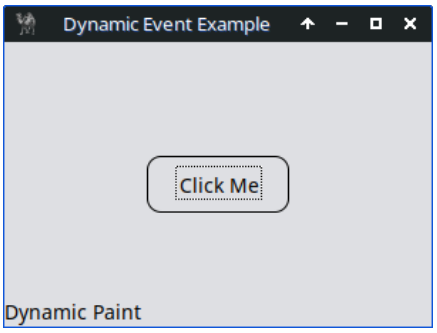


Figure 21.2: Dynamic Event Handler Registration

```
use Prima qw(Application Buttons);

my %event_handlers = (
    on_click => sub {
        my ($button) = @_;
        $button->text("I Was Clicked!");  # Clearer text change
        $button->color(cl::LightGreen);   # Visual feedback
        print "Button clicked! Text updated.\n";
    },
    on_paint => sub {
        my ($window, $canvas) = @_;
        $canvas->clear;
        $canvas->text_shape_out("Dynamic Paint", 0, 0);
    },
);

my $mw = Prima::MainWindow->new(
    text    => 'Dynamic Event Example',
    size    => [300, 200],
    icon   => Prima::Icon->load('icon.png'),
    onPaint => $event_handlers{on_paint},
);

my $button = $mw->insert(
    Button =>
        text   => 'Click Me',  # Initial text
        origin => [100, 80],
        size   => [100, 40],
        onClick => $event_handlers{on_click},
);

Prima->run;
```

Listing 21.2: Dynamic Event Handler Registration

Summary key differences between the two examples

Feature	Modular Event Handlers	Dynamic Event Handler Registration
Handler Storage	Standalone subroutines	Centralized in a hash
Assignment	Static (`\&handler_name`)	Dynamic (`$hash{handler}`)
Flexibility	Fixed at creation	Can change at runtime

Feature	Modular Event Handlers	Dynamic Event Handler Registration
Use Case	Clean, maintainable code	Adaptive, runtime-configurable behavior
Example Change	Requires code edit	Update hash: `$hash{on_click} = sub {...}`

This is a perfect starting point for interactive UIs where controls react visibly. To make your life easier, here's a list of common Prima widget event handlers (like `onPaint`, `onClick`, etc.) that you can use in simple programs like the examples you've been building.

These handlers are often defined as widget properties (e.g. `onClick => sub { ... }` or `onClick => \&handler`). Many apply to most visual widgets.

Widget Type	Common Event Handlers	Notes
`Prima::MainWindow`	`onPaint`, `onSize`, `onMove`, `onClose`, `onKeyDown`, `onKeyUp`, `onMouseDown`, `onMouseUp`	Top-level application window
`Prima::Button`	`onClick`, `onMouseDown`, `onMouseUp`, `onPaint`	Simple clickable button
`Prima::InputLine`	`onChange`, `onKeyDown`, `onKeyUp`, `onFocus`, `onPaint`	Single-line text input
`Prima::Edit`	`onChange`, `onKeyDown`, `onKeyUp`, `onMouseDown`, `onPaint`	Multi-line text box
`Prima::Label`	`onPaint`, `onMouseDown`, `onMouseUp`	Passive text label
`Prima::ListBox`	`onSelectItem`, `onDrawItem`, `onMouseDown`, `onPaint`	Scrollable item list
`Prima::ComboBox`	`onSelectItem`, `onChange`, `onDrawItem`	Editable dropdown
`Prima::Timer`	`onTick` (or `onTimer`)	Fires periodically when enabled
`Prima::Widget`	`onPaint`, `onMouseDown`, `onMouseUp`, `onMouseMove`, `onKeyDown`, `onKeyUp`, `onEnter`, `onLeave`, `onFocus`, `onSize`, `onMove`	

Table 21.1 Prima Widget Event Handler Cheat Sheet

Handler	Triggered When…
`onClick`	The user clicks (usually a button or clickable widget)
`onChange`	The widget’s content changes (e.g., text updated)
`onPaint`	The widget needs to redraw itself
`onSize`	The widget is resized
`onMove`	The widget is moved
`onKeyDown`	A key is pressed
`onKeyUp`	A key is released
`onMouseDown`	A mouse button is pressed inside the widget
`onMouseUp`	A mouse button is released inside the widget
`onMouseMove`	The mouse moves over the widget
`onEnter`	The mouse enters the widget’s area
`onLeave`	The mouse leaves the widget’s area
`onClose`	A window is closing (you may cancel the close)
`onFocus`	The widget receives keyboard focus
`onSelectItem`	A selection changes (e.g., list box, combo box)

Handler	Triggered When…
onDrawItem	The widget draws a single item (e.g., in a list)
onTimer	A timer event fires

Table 21.2 Event Handler Descriptions (Quick Reference)

Notes

Widget-specific handlers (e.g., *onSelectItem* for ListBox, *onChange* for InputLine) only apply to certain classes. You can check the Prima widget documentation for a full list per widget.

All handlers follow the naming convention *onEventName*, and you assign them like:

```
$widget->onClick( sub { ... } );
```

or:

```
onClick => \&handler_function
```

21.1.1 Practical example

The program displays a list of countries -like Slovakia, Hungary, and Poland - with their populations, formatted clearly between two static lines. The total population is also shown.

For the static lines we define an *onPaint* function:

```
# Draw horizontal lines
$mw->onPaint(sub {
    my ($self) = @_;
    $self->clear;
    $self->color(cl::Black);

    $self->line($C{H_MARGIN}, $UPPER_LINE_Y,  $C{WINDOW_WIDTH} - $C{H_MARGIN},
                                              $UPPER_LINE_Y);
    $self->line($C{H_MARGIN}, $LOWER_LINE_Y,  $C{WINDOW_WIDTH} - $C{H_MARGIN},
                                              $LOWER_LINE_Y);
});
```

The 16 label items are defined simply using a *for* loop that processes the *@countries* array.

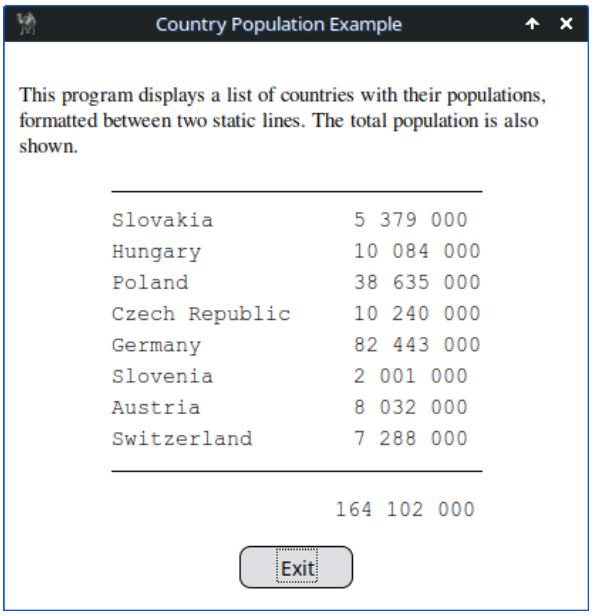


Figure 21.3: Country Population Example with Static Lines

```
use Prima qw(Application Label Buttons);
use List::Util qw(sum);

# Data
my @countries = (
    ['Slovakia',       '5 379 000'],
    ['Hungary',        '10 084 000'],
    ['Poland',         '38 635 000'],
    ['Czech Republic', '10 240 000'],
    ['Germany',        '82 443 000'],
    ['Slovenia',       '2 001 000'],
    ['Austria',        '8 032 000'],
    ['Switzerland',    '7 288 000'],
);

# Constants
my %C = (
    WINDOW_HEIGHT      => 400,
    WINDOW_WIDTH       => 410,
    TOP_MARGIN         => 25,
    BOTTOM_MARGIN      => 25,
    LINE_SPACING       => 22,
    DESCRIPTIVE_HEIGHT => 60,
    H_MARGIN           => 75,
);

# Total population calculation
my $total_population =
    sum map { s/\s+//gr } map { $_->[1] } @countries;

my $formatted_total = join ' ',
    unpack 'A3 A3 A3', $total_population;

# Shared label style
my %label_style = (
    font  => { name => 'Courier New', size => 11 },
    color => cl::Black,
);

# Main window
my $mw = Prima::MainWindow->new(
    text        => 'Country Population Example',
    size        => [$C{WINDOW_WIDTH}, $C{WINDOW_HEIGHT}],
    icon        => Prima::Icon->load('icon.png'),
    backColor   => cl::White,
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu | bi::TitleBar,
);

# Exit button
$mw->insert(
    Button =>
        origin  => [$C{WINDOW_WIDTH} / 2 - 40, $C{BOTTOM_MARGIN} - 10],
        size    => [80, 30],
        text    => 'Exit',
        onClick => sub { $mw->close },
);

# Total population label
my $TOTAL_POP_Y = $C{BOTTOM_MARGIN} + 30;
```

```
$mw->insert(
    Label =>
        origin => [232, $TOTAL_POP_Y],
        text   => $formatted_total,
        %label_style,
);

# Descriptive label
my $DESCRIPTION_Y = $C{WINDOW_HEIGHT} - $C{TOP_MARGIN} -
                                        $C{DESCRIPTIVE_HEIGHT};

$mw->insert(
    Label =>
        origin      => [10, $DESCRIPTION_Y],
        width       => $C{WINDOW_WIDTH} - 20,
        text        =>
            "This program displays a list of countries with their " .
            "populations, formatted between two static lines. The " .
            "total population is also shown.",
        wordWrap    => 1,
        autoHeight  => 1,
        borderWidth => 1,
        font        => { name => 'Times', size => 11 },
        color       => cl::Black,
);

# Calculate positions
my $LAST_COUNTRY_Y  = $TOTAL_POP_Y + 50;
my $FIRST_COUNTRY_Y = $LAST_COUNTRY_Y + $C{LINE_SPACING} * (@countries - 1);

my $UPPER_LINE_Y = $FIRST_COUNTRY_Y + 35;
my $LOWER_LINE_Y = $LAST_COUNTRY_Y + $C{LINE_SPACING} - 30;

# Draw horizontal lines
$mw->onPaint(sub {
    my ($self) = @_;
    $self->clear;
    $self->color(cl::Black);

    $self->line($C{H_MARGIN}, $UPPER_LINE_Y,  $C{WINDOW_WIDTH} - $C{H_MARGIN},
                                              $UPPER_LINE_Y);
    $self->line($C{H_MARGIN}, $LOWER_LINE_Y,  $C{WINDOW_WIDTH} - $C{H_MARGIN},
                                              $LOWER_LINE_Y);
});

# Country labels (bottom -> top)
my $y = $LAST_COUNTRY_Y;

for my $country (reverse @countries) {
    my ($name, $pop) = @$country;

    $mw->insert(
        Label =>
            origin => [$C{H_MARGIN}, $y],
            text   => $name,
            %label_style,
    );

    $mw->insert(
        Label =>
            origin => [245, $y],
            text   => $pop,
            %label_style
```

```
            %label_style,
    );

    $y += $C{LINE_SPACING};
}

Prima->run;
```

Listing 21.3: Country Population Example with Static Lines

21.2 Extension strategy 2: Subclassing and Overriding Methods

Subclassing is a powerful way to extend functionality. In the next example, we'll define our own widget by inheriting from a built-in Prima class and overriding a method to change how it behaves.

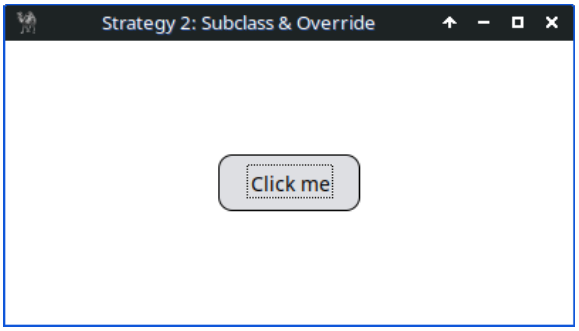


Figure 21.4: subclassing and overriding methods

```
use Prima qw(Application Buttons);

# Define a custom button class MyButton that inherits from Prima::Button
package MyButton;
use base 'Prima::Button';

# Override the profile_default method to customize default widget properties
sub profile_default {
    # Get the object invocant
    my ($self) = @_;

    # Call parent class profile_default to get base defaults
    my $def = $self->SUPER::profile_default;

    # Set a custom onClick event handler for this button
    $def->{onClick} = sub {
        # Event handler arguments: widget and button clicked
        my ($self, $btn) = @_;
        # Print a message to the console when the button is clicked
        print "Button clicked\n";
    };
    return $def; # Return modified profile hashref
}

# Define a custom canvas class MyCanvas that inherits from Prima::Widget
package MyCanvas;
use base 'Prima::Widget';

# Override the profile_default method to customize default widget properties
sub profile_default {
    my ($self) = @_;
    my $def = $self->SUPER::profile_default;

    # Set a custom onMouseDown event handler for this canvas
    $def->{onMouseDown} = sub {
        # Event handler args: widget, mouse button, modifier keys, x, y coords
        my ($self, $btn, $mod, $x, $y) = @_;
        # Print coordinates when canvas is clicked
        print "Canvas clicked at ($x, $y)\n";
    };
    return $def;
}

# Return to the main package for script execution
package main;

# Create the main application window
my $mw = Prima::MainWindow->new(
    text   => 'Strategy 2: Subclass & Override',
    size   => [400, 200],
    icon  => Prima::Icon->load('icon.png'),
);

# Insert a MyCanvas widget into the main window, covering the whole window
my $canvas = $mw->insert('MyCanvas',
    origin    => [0, 0], # Position top-left corner
    size      => [$mw->size],
    backColor => cl::White,
);

# Insert a MyButton widget into the canvas at a specified location
```

```
$canvas->insert('MyButton',
    origin => [150, 80],
    size   => [100, 40],
    text   => 'Click me',
);

Prima->run;
```

Listing 21.4: Subclassing and Overriding Methods

Output e.g.

```
Canvas clicked at (340, 129)
Button clicked
Canvas clicked at (40, 24)
Button clicked
```

Should I use this approach?

Pros:

- Ideal for reusable components or large applications.
- Promotes clean, object-oriented design.

Cons:

- Adds boilerplate code.
- May be unnecessary for simple or one-off behaviors.

For simpler cases, inline event handlers may be sufficient (see Section 21.1).

The above example demonstrates the basics of subclassing and overriding the `profile_default` method to customize widget behavior. While this is a good start, the example is somewhat minimal and could be expanded to better illustrate the power and flexibility of subclassing in Prima. Let's find out with three concise examples.

21.2.1 Custom Button with Enhanced Functionality

Show how to create a custom button that not only logs a message but also changes its appearance (e.g., color, text) when clicked. This will demonstrate how to override multiple methods and properties for more complex behavior.

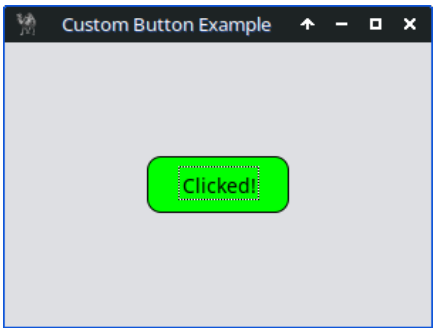


Figure 21.5: Custom Button with Enhanced Functionality

```
use Prima qw(Application Buttons);

# Define a custom button class
package MyCustomButton;
use base 'Prima::Button';

# Override profile_default to set default properties
sub profile_default {
    my ($self) = @_;
    my $def = $self->SUPER::profile_default;
    $def->{text} = 'Click Me!';
    $def->{backColor} = cl::LightGray;
    $def->{onClick} = sub {
        my ($self) = @_;
        $self->text('Clicked!');
        $self->backColor(cl::LightGreen);
        print "Button was clicked!\n";
    };
    return $def;
}

# Main application
package main;
my $mw = Prima::MainWindow->new(
    text => 'Custom Button Example',
    size => [300, 200],
    icon => Prima::Icon->load('icon.png'),
);

# Insert the custom button
$mw->insert('MyCustomButton',
    origin => [100, 80],
    size => [100, 40],
);

Prima->run;
```

Listing 21.5: Custom Button with Enhanced Functionality

21.2.2 Custom Input Field with Validation

Create a custom input field that validates user input (e.g., only allows numeric input). This will show how to override methods like `onKeyDown` to enforce custom logic.

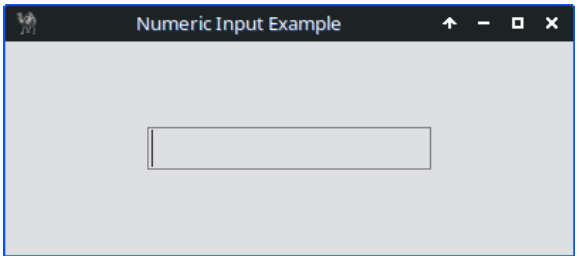


Figure 21.6: Custom Input Field with Numeric Validation

```
use Prima qw(Application InputLine);

package NumericInput;
use base 'Prima::InputLine';

sub profile_default {
    my ($self) = @_;
    my $def = $self->SUPER::profile_default;
    # Set the initial text to empty
    $def->{text} = '';
    $def->{onChange} = sub {
        my ($self) = @_;
        my $text = $self->text;
        # Remove all non-digit characters
        $text =~ s/[^\d]//g;
        # Only update if the text changed
        if ($text ne $self->text) {
            $self->text($text);
        }
    };
    return $def;
}

package main;
my $mw = Prima::MainWindow->new(
    text => 'Numeric Input Example',
    size => [400, 150],
    icon => Prima::Icon->load('icon.png'),
);

# Define the width of the input field
my $input_width = 200;

# Insert the custom input field and center it
my $input = $mw->insert(
    'NumericInput',
    origin => [($mw->width - $input_width) / 2, 60],
    size   => [$input_width, 30],
    growMode => gm::Center,
);

Prima->run;
```

Listing 21.6: Custom Input Field with Numeric Validation

21.2.3 Custom Canvas with Drawing Functionality

In this larger example we create our own canvas widget by subclassing `Prima::Widget`. The widget demonstrates overriding event methods to handle drawing:

- `on_paint` - redraw all stored line segments whenever the canvas repaints.
- `on_mousedown` - start drawing and record the initial mouse position.
- `on_mousemove` - draw new line segments while the mouse moves.
- `on_mouseup` - stop drawing.

The example also shows how to use Prima's built-in tools to enhance the interface:

- ColorDialog - to pick the line color.
- Slider - to adjust line thickness.
- Button - to clear the canvas.

By combining subclassing with Prima's built-in widgets, the program is clean, reusable, and interactive, demonstrating the core patterns for GUI development in Prima.

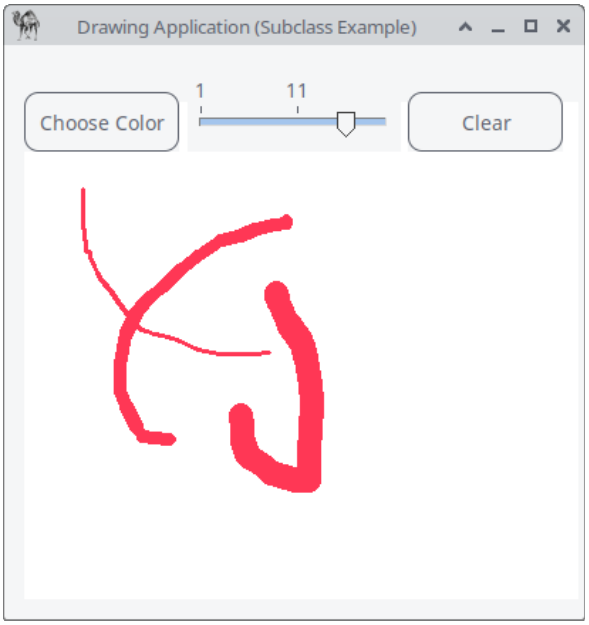


Figure 21.7: Custom Canvas with Drawing Functionality

```
use Prima qw(Application Buttons Dialog::ColorDialog Sliders);

# ============================================================
#  MyCanvas – a custom drawing widget
#  Demonstrates subclassing and overriding Prima widget methods
# ============================================================

package MyCanvas;
use base qw(Prima::Widget);

sub init {
    my ($self, %profile) = @_;
    $self->SUPER::init(%profile);

    # internal state
    $self->{segments}          = [];
    $self->{drawing}           = 0;
    $self->{last_x}            = 0;
    $self->{last_y}            = 0;
    $self->{current_color}     = cl::Black;
    $self->{current_linewidth} = 2;

    return $self;
}

# ------------------------------------------------------------
# Overridden paint event: redraw all stored line segments
# ------------------------------------------------------------
sub on_paint {
    my ($self, $canvas) = @_;

    $canvas->clear;

    for my $s (@{ $self->{segments} }) {
        $canvas->color( $s->{color} );
        $canvas->lineWidth( $s->{width} );
        $canvas->line( @{ $s->{points} } );
    }
}

# ------------------------------------------------------------
# Overridden mouse-down event: start drawing
# ------------------------------------------------------------
sub on_mousedown {
    my ($self, $btn, $mod, $x, $y) = @_;
    $self->{drawing} = 1;
    $self->{last_x}  = $x;
    $self->{last_y}  = $y;
}

# ------------------------------------------------------------
# Overridden mouse-move event: draw while button is held
# ------------------------------------------------------------
sub on_mousemove {
    my ($self, $mod, $x, $y) = @_;
    return unless $self->{drawing};

    push @{ $self->{segments} }, {
        color  => $self->{current_color},
        width  => $self->{current_linewidth},
        points => [ $self->{last_x}, $self->{last_y}, $x, $y ],
```

```
    };

    $self->{last_x} = $x;
    $self->{last_y} = $y;

    $self->repaint;
}

# ------------------------------------------------------------
# Overridden mouse-up event: stop drawing
# ------------------------------------------------------------
sub on_mouseup {
    my ($self) = @_;
    $self->{drawing} = 0;
}

# ------------------------------------------------------------
# Helper method: clear the drawing
# ------------------------------------------------------------
sub clear_canvas {
    my ($self) = @_;
    $self->{segments} = [];
    $self->repaint;
}

# ------------------------------------------------------------
# Helper method: change current color
# ------------------------------------------------------------
sub set_color {
    my ($self, $color) = @_;
    $self->{current_color} = $color;
}

# ------------------------------------------------------------
# Helper method: change current line width
# ------------------------------------------------------------
sub set_linewidth {
    my ($self, $width) = @_;
    $self->{current_linewidth} = $width;
}

# ============================================================
#  Main Program
# ============================================================

package main;

my $mw = Prima::MainWindow->new(
    text => 'Drawing Application (Subclass Example)',
    size => [400, 400],
    icon => Prima::Icon->load('icon.png'),
);

# Create our custom canvas
my $canvas = MyCanvas->new(
    owner  => $mw,
    origin => [10, 10],
    size   => [580, 350],
    backColor => cl::White,
);

# ---- COLOR PICKER BUTTON ----
```

```
$mw->insert( Button =>
        text   => 'Choose Color',
        origin => [10, 325],
        onClick => sub {
            my $dlg = Prima::Dialog::ColorDialog->new;
            if ($dlg->execute) {
                $canvas->set_color( $dlg->value );
            }
        },
);

# ---- LINE WIDTH SLIDER ----
$mw->insert( Slider =>
        origin => [125, 325],
        width  => 150,
        min    => 1,
        max    => 20,
        value  => 2,
        onChange => sub {
            $canvas->set_linewidth( $_[0]->value );
        },
);

# ---- CLEAR CANVAS BUTTON ----
$mw->insert( Button =>
        text   => 'Clear',
        origin => [280, 325],
        onClick => sub {
            $canvas->clear_canvas;
        },
);

Prima->run;
```

Listing 21.7: Custom Canvas with Drawing Functionality

Remember:

- Use `profile_default` for *default properties,*
- use `init` when your widget requires internal state (like arrays, flags, custom attributes).

21.3 Extension strategy 3: Using Composition (Has-a Relationship)

This strategy is about designing own objects that contain Prima objects instead of inheriting from them. Composition means that one object contains another. This is known as a has-a relationship, e.g. a Toolbar has many buttons. This is different from inheritance (is-a relationship), where one class extends another. A simple example:

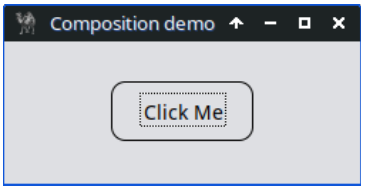


Figure 21.8 Composition in Prima – Simple Controller Example

```
use Prima qw(Application Buttons);

package MyController;

# Constructor for the controller class
sub new {
    my ($class, $parent) = @_;  # Expect a parent window
    my $self = bless {}, $class;

    # Create a button as part of the controller, assign to the parent window
    $self->{button} = Prima::Button->new(
        owner => $parent, # Attach to the given window
        text  => 'Click Me',
        left  => 75,
        bottom => 30,
        width => 100,
        onClick => sub {
            print "Handled by controller\n";
        },
    );

    return $self;
}

package main;

my $mw = Prima::MainWindow->new(
    text => 'Composition demo',
    size => [250, 100],
    icon  => Prima::Icon->load('icon.png'),
);

# Instantiate the controller and pass the window as its parent
my $controller = MyController->new($mw);

Prima->run;
```

Listing 21.8: Composition in Prima – Simple Controller Example

Two more examples.

21.3.1 A StatusBar object that contains labels

In this example, we demonstrate composition in Prima by creating a `StatusBar` object that *has two labels*. The `StatusBar` itself is not a label — instead, it contains `Prima::Label` objects to display information on the left and right sides of a window. This "has-a" relationship clearly illustrates how one object can manage multiple components without inheriting from them.

By organizing the labels inside the `StatusBar`, we separate the responsibilities: the main window doesn't need to know about individual labels, and the `StatusBar` can control its own layout and content.

Figure 21.9 Composition, StatusBar Example

```
use Prima qw(Application Label);

package StatusBar;

sub new {
    my ($class, $parent) = @_;
    my $self = bless {}, $class;

    # Left label
    $self->{left_label} = Prima::Label->new(
        owner  => $parent,
        text   => 'Ready',
        left   => 5,
        bottom => 5,
        width  => 100,
    );

    # Right label
    $self->{right_label} = Prima::Label->new(
        owner  => $parent,
        text   => 'Line 1',
        left   => $parent->width - 105,  # 5px margin from right
        bottom => 5,
        width  => 100,
        alignment => ta::Right,
    );

    return $self;
}

package main;

my $mw = Prima::MainWindow->new(
    text => 'StatusBar demo',
    size => [300, 100],
    icon => Prima::Icon->load('icon.png'),
);

my $status = StatusBar->new($mw);

Prima->run;
```

Listing 21.9: Composition, StatusBar Example

21.3.2 A Toolbar object that contains multiple buttons

This example demonstrates composition (“has-a”) in Perl’s Prima GUI framework. We define a `Toolbar` object that has a panel (`Prima::Widget::Panel`) and that panel has multiple buttons (`Prima::Button`).

The key idea is that the `Toolbar` itself is not a button or a panel — it contains them, showing how objects can manage other objects without using inheritance.

The panel acts as a container at the top of the main window, and the buttons are positioned relative to it. An `onSize` handler ensures the panel always spans the full window width when resized, keeping the toolbar visually consistent.

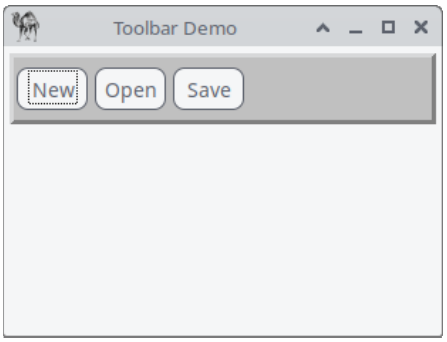


Figure 21.10 Composition, Toolbar Example

```
use Prima qw(Application Buttons);
use Prima::Widget::Panel;

package Toolbar;

sub new {
    # $class = 'Toolbar', $parent = main window
    my ($class, $parent) = @_;
    # Create a new object (hash) and bless it into Toolbar class
    # $self will *have* panel and buttons — demonstrates composition ("has-a")
    my $self = bless {}, $class;

    my $panel_height = 50;

    # Create the panel at the top of the window
    $self->{panel} = Prima::Widget::Panel->new(
        owner       => $parent, # Panel is inside the main window
        left        => 0,
        bottom      => $parent->height - $panel_height - 5,
        width       => $parent->width,
        height      => $panel_height,
        borderWidth => 3, # Optional: give a border for visibility
        color       => cl::LightGray,
        backColor   => cl::LightGray,
    );

    # Buttons inside the panel
    # These are *owned by the panel*, demonstrating composition:
    # Toolbar has-a Panel, Panel has Buttons
    $self->{buttons} = [
        Prima::Button->new( owner => $self->{panel}, text => 'New',
                            left => 5,  bottom => 10,
                            width => 50, height => 30),
        Prima::Button->new( owner => $self->{panel}, text => 'Open',
                            left => 60, bottom => 10,
                            width => 50, height => 30),
        Prima::Button->new( owner => $self->{panel}, text => 'Save',
                            left => 115,bottom => 10,
                            width => 50, height => 30),
    ];

    return $self; # Return the fully constructed Toolbar object
}

package main;

# Create the main application window
my $mw = Prima::MainWindow->new(
    text => 'Toolbar Demo',
    size => [300, 200],
    icon => Prima::Icon->load('icon.png'),
);

# Instantiate the Toolbar, passing the main window as its parent
my $toolbar = Toolbar->new($mw);

# Make the panel span full width and stay at top when window resizes
$mw->onSize(sub {
    $toolbar->{panel}->width($mw->width);          # Update panel width
    $toolbar->{panel}->bottom($mw->height - 50);  # Keep panel at top
});
```

```
Prima->run;
```

Listing 21.10: Composition, Toolbar Example

21.4 Extension strategy 4: Mixin Modules (Sharing Code Between Classes)

A mixin module is a lightweight, reusable piece of code that provides shared behavior - like logging or formatting - that can be used by multiple classes without relying on inheritance. It's not a class, but a standalone module containing functions that other classes can call. This approach helps you avoid duplicating code, makes your design more flexible, and keeps things organized by separating common functionality from your main logic.

I made a simple demo illustrating the use of mixins to add reusable behavior to buttons and shared functionality, such as logging or tooltips, without using inheritance.

The demo creates a main window with three buttons demonstrating different combinations of mixins:

1. **Log Only** – prints a message to the console when clicked.
2. **Tooltip Only** – shows a tooltip when hovered.
3. **Log + Tooltip** – combines both behaviors.

Using mixins keeps your code modular and avoids duplicating functionality across multiple classes.

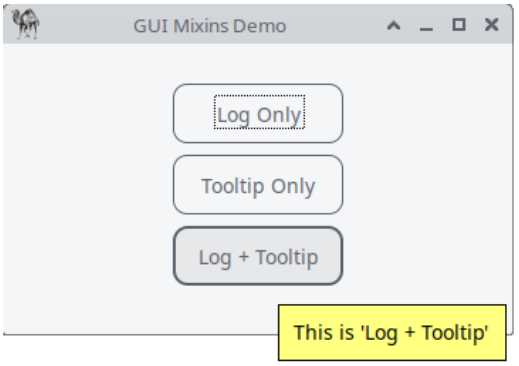


Figure 21.11: Mixin Modules, Reusable Behavior

```
use Prima qw(Application Buttons Label);

# --- Mixins ---

# LoggerMixin: provides reusable logging behavior
package LoggerMixin;
sub log {
    my ($msg) = @_;
    print "[LOG] $msg\n";    # Print a message to the console
}

# TooltipMixin: provides simple tooltip behavior
package TooltipMixin;
sub set_tooltip {
    my ($button, $tip) = @_;
    my $mw = $button->owner;  # reference to parent window (not used here)

    # Set the tooltip for the button when hovered
    $button->onMouseEnter(sub {
        $button->hint($tip);  # Prima automatically shows this as a tooltip
    });
}

# --- Button Controller Class ---

package MyButton;
sub new {
    my ($class, $parent, $text, $left, $bottom, $apply_logging,
                                                $apply_tooltip) = @_;
    my $self = bless {}, $class;

    # Create the actual Prima button
    $self->{button} = Prima::Button->new(
        owner  => $parent,
        text   => $text,
        left   => $left,
        bottom => $bottom,
        width  => 120,
    );

    # Apply logging behavior if requested
    if ($apply_logging) {
        $self->{button}->onClick(sub { LoggerMixin::log("Button '$text' clicked") });
    }

    # Apply tooltip behavior if requested
    if ($apply_tooltip) {
        TooltipMixin::set_tooltip($self->{button}, "This is '$text'");
    }

    return $self;
}

# --- Main Application ---

package main;

# Create the main window
my $mw = Prima::MainWindow->new(
    text => 'GUI Mixins Demo',
    size => [350, 200],
```

```
);

# Instantiate buttons with different combinations of mixins
MyButton->new($mw, 'Log Only', 50, 120, 1, 0);       # logging only
MyButton->new($mw, 'Tooltip Only', 50, 80, 0, 1);    # tooltip only
MyButton->new($mw, 'Log + Tooltip', 50, 40, 1, 1);   # both logging and tooltip

Prima->run;
```

Listing 21.11: Mixin Modules, reusable behavior

21.5 When to Use Which Strategy?

Each extension technique from this chapter serves a different purpose. Here is a quick guide:

- **Event Callbacks**
  Use callbacks when you only need to react to user actions or redraw events, and the behavior is simple. Ideal for small programs or when the logic naturally belongs in the main script.
- **Subclassing**
  Choose subclassing when you want to create reusable custom widgets or when a widget needs its own internal state, drawing logic, or default properties. This approach works best for larger or more structured applications.
- **Composition (Has-a)**
  Use composition when your object *manages* other widgets—such as toolbars, status bars, or controllers—without becoming a widget itself. This keeps responsibilities clean and avoids deep inheritance trees.
- **Mixins**
  Mixins are helpful when several classes need the same small piece of behavior, such as logging, formatting, or tooltips. They reduce code duplication without forcing inheritance relationships.

Closing words

Extending Prima is ultimately about choosing the approach that keeps your code clear, expressive, and easy to maintain. Whether you attach a simple callback, build a reusable widget through subclassing, organize your interface using composition, or share behavior through mixins, each technique gives you a different level of control over how your application behaves.

The more you work with Prima, the more naturally these strategies will fit together. Most real programs combine them: callbacks for quick reactions, subclasses for custom components, composition for structural clarity, and mixins for shared functionality. Knowing when to apply each one is what turns small scripts into well-designed applications.

Part 9 – Menus and Building Larger Applications

22. Menus and Application Structure

Menus are one of the most important interaction tools in a Prima application. They organize commands, expose application features, and can adjust dynamically at runtime. This chapter explains menu construction step by step - from basic menu syntax to icons, dynamic updates, MRU lists, and finally their integration into a functional text editor.

22.1 Building Menus the Right Way

Prima menus are defined using array references whose structure depends on the number of elements provided. While small arrays work for simple cases, the six-element format gives you the most control and is the recommended form for production applications.

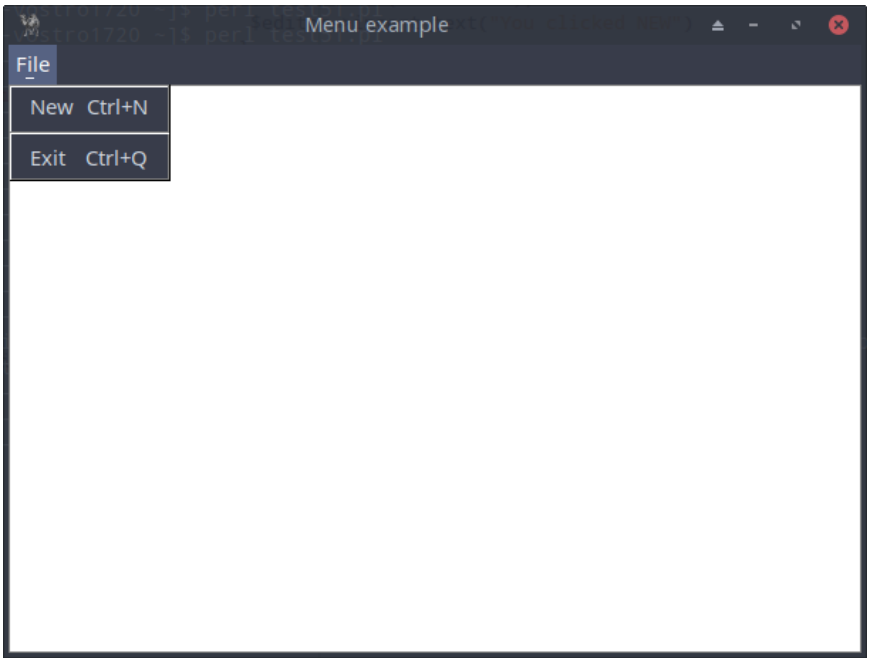


Figure 22.1: Basic Menu

Let’s explore the foundational steps for creating menus.

1. Adding the "File" Menu

A minimal menu bar begins with a top-level item such as File:

```
[ 'F~ile' => [ ]

],
```

`'F~ile'`

- This is the label for the menu item.

- The ~ character specifies a keyboard shortcut (Alt-accelerator key), here Alt + I (since Alt + F might conflict with another system shortcut on some devices). Without the ~, the label would appear as "File" without a keyboard shortcut.

`[ ]`

- This is an empty array reference, indicating there are no submenu items under File yet. It acts as a placeholder for submenus to be added later.

2. Adding Submenus

A typical File menu includes entries such as New, Exit, and a separator:

```
[ 'F~ile' => [
                [ 'New', 'Ctrl+N', '^N', sub { } ],
                [], # separator line
                [ 'Exit', 'Ctrl+Q', '^Q', sub { exit } ],
            ]
],
```

Here's what each component does:

['New', 'Ctrl+N', '^N', sub { }]

- Adds a New menu item.

- Includes Ctrl+N or ^N which represent the displayed and actual keyboard accelerators.

- Assigns an empty callback function: sub { }, which you can later replace with your desired functionality.

[]

- Represents a separator line between menu items.

['Exit', 'Ctrl+Q', '^Q', sub { exit }]

- Adds an Exit menu item.

- Includes displayed and actual keyboard accelerators: Ctrl+Q or ^Q.

- Executes the exit function to close the application.

Prima also supports a shortcut-style syntax: you can define Exit as *["E~xit" => "Exit"]*, enabling the *Alt + X*. But for clarity and consistency, the full array form is recommended.

22.2 Icons, Dynamic Menus, and Recent- Files Lists

22.2.1 Submenu icons

Submenu items are constructed using menu description arrays, which can contain up to six elements. While arrays with fewer elements are interpreted differently, the most detailed format - a six-scalar array - provides a fully qualified text-item description (and is all you need!). The format for this array is as follows:

[NAME, TEXT/IMAGE, ACCEL, KEY, ACTION/SUBMENU, DATA]

An example with icons:

```
[ 'F~ile' =>
  [

    # [ NAME, TEXT/IMAGE, ACCEL, KEY, ACTION/SUBMENU, DATA ]
    [ 'New', ' New', 'Ctrl+N', '^N', sub { print("new\n");  },
                                                 { icon => $icon_new } ],
    [],
    # space in second field is necessary to separate icon and text
    [ 'Open', ' Open', 'CTRL+O', '^O', sub { print("open\n");  },
                                                { icon => $icon_open } ],
    [ 'Save', ' Save', 'CTRL+S', '^S', sub { print("save\n");  },
                                                { icon => $icon_save } ],
    [],
    [ 'Exit', ' Exit', 'Ctrl+Q', '^Q', sub { exit },
                                                { icon => $icon_quit } ],
  ]
],
```

Listing 22.1: Menu and Submenu Items

The fields here are:

- NAME: the internal identifier for the menu item, e.g., 'Open'.
- TEXT: the caption displayed in the menu, e.g., " Open".
- ACCEL: a description of the shortcut for display purposes, e.g., Ctrl+O.
- KEY: the key binding for the shortcut, where ^ represents the Ctrl key, e.g., ^O.

- SUBMENU: the action or subroutine to execute when the menu item is selected, e.g., `sub { print("open\n"); }`.
- DATA: additional data, such as an icon reference, e.g., `{ icon => $icon_open }`.

The icons are loaded using the `Prima::Icon` module:

```
my $icon_new = Prima::Icon->load('new.png');
my $icon_open = Prima::Icon->load('open.png');
my $icon_save = Prima::Icon->load('save.png');
my $icon_quit = Prima::Icon->load('quit.png');
```

The code of the next application should be fully clear now.

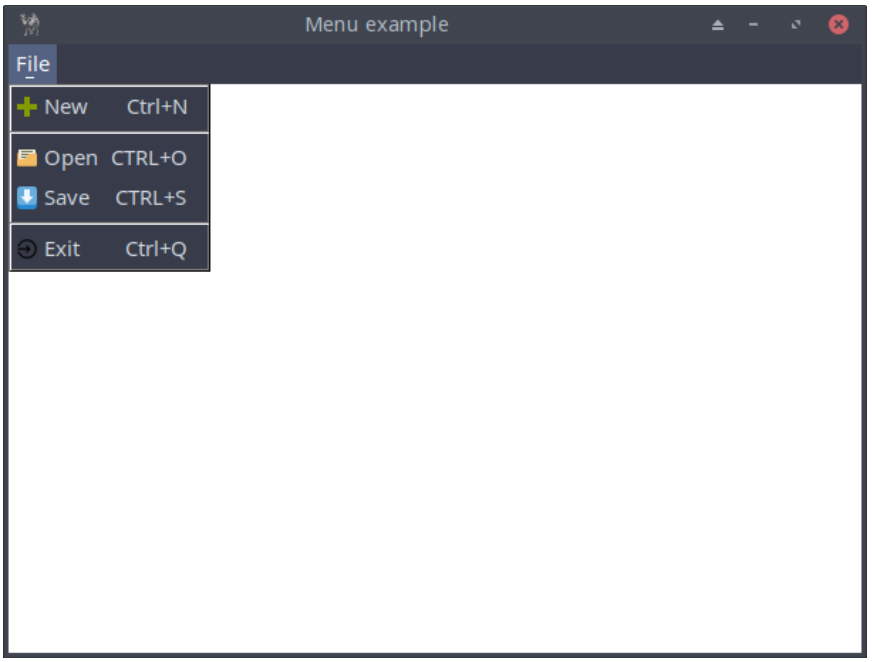


Figure 22.2: Submenu Icons

```
use Prima qw(Application);

use lib '.';

my $icon_new = Prima::Icon->load('new.png');
my $icon_open = Prima::Icon->load('open.png');
my $icon_save = Prima::Icon->load('save.png');
my $icon_quit = Prima::Icon->load('quit.png');

my $mw = Prima::MainWindow->new(
    text      => "Menu example",
    size      => [600, 400],
    backColor => cl::White,
    icon  => Prima::Icon->load('icon.png'),
    # menuFont => { name => 'Courier New', size => 11, },
    menuItems => [
    [ 'F~ile' =>
    [

      # [ NAME, TEXT/IMAGE, ACCEL, KEY, ACTION/SUBMENU, DATA ]
      [ 'New', ' New', 'Ctrl+N', '^N', sub { print("new\n");  },
                                                 { icon => $icon_new } ],
      [],
      # space necessary to  separate icon and text
      [ 'Open', ' Open', 'CTRL+O', '^O', sub { print("open\n");  },
                                                { icon => $icon_open } ],
      [ 'Save', ' Save', 'CTRL+S', '^S', sub { print("save\n");  },
                                                { icon => $icon_save } ],
      [],
      [ 'Exit', ' Exit', 'Ctrl+Q', '^Q', sub { exit },
                                                { icon => $icon_quit } ],
      ]
    ],
);

Prima->run;
```

Listing 22.2: Submenu Icons

If you use a monospaced menu font. e.g.

```
menuFont => { name => 'Courier', size => 11, },
```

then the menu items are perfectly right-aligned.

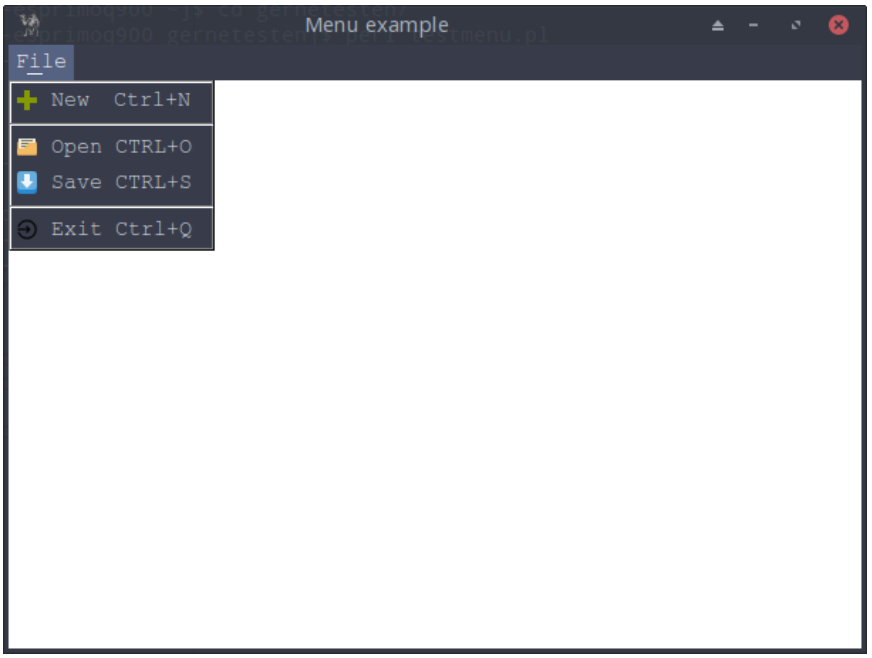


Figure 22.3: Submenu Icons Right Aligned

Examples of monospaced fonts on Linux include:

- Courier (Standard)
- DejaVu Sans Mono
- Ubuntu Mono
- Monospace (Generic fallback font)
- Droid Sans Mono

and they are definitely worth trying out.

22.2.2 Dynamic menus and Recent-Filelist

Dynamic menus are about changing menu structure at runtime, while a Recent-Files list - which is simply a special case of a dynamic menu- uses that mechanism to display a changing history of recently opened files.

Here an example of a dynamic menu: an update checker which could be found in the Help menu. When the application starts, the menu may contain 'Check for updates...' as in:

```
['ID10','Check for updates…' => sub { check_for_updates(); }],
```

If the program later discovers that a new version is available, the label can be updated instantly: 'Download version 2.0.1.'

In Prima this is done by modifying the menu structure and reassigning it:

```
sub check_for_updates {

    # ------------------------------------------------
    # Step 1: Simulate discovering a new version
    # ------------------------------------------------
    # Pretend we checked online and found version 2.0.1
    my $new_version = "2.0.1";

    # ------------------------------------------------
    # Step 2: Loop through all top-level menus
    # $menu_items contains arrays like ['F~ile', [...]], ['Hel~p', [...]]
    # ------------------------------------------------
    foreach my $menu (@$menu_items) {

        # Skip everything except the Help menu
        #   - $menu->[0] is the menu name ('F~ile', 'Hel~p', etc.)
        #   - $menu->[1] should be an arrayref with all submenu items
        next unless $menu->[0] eq 'Hel~p' && ref($menu->[1]) eq 'ARRAY';

        # ------------------------------------------------
        # Step 3: Loop through all items inside Help menu
        # ------------------------------------------------
        foreach my $item (@{$menu->[1]}) {

            # Skip separators or any item without an ID
            # - ref($item) eq 'ARRAY' ensures it's a proper menu item
            # - defined $item->[0] ensures it has an ID
            if (ref($item) eq 'ARRAY' && defined $item->[0] &&
                                                 $item->[0] eq 'ID10') {

                # ------------------------------------------------
                # Step 4: Found our "Check for updates" menu item
                # Change its label to show the new version
                # ------------------------------------------------
                $item->[1] = "Download version $new_version";

                # Optional console feedback
                print "New version found! Menu updated.\n";
                # Stop looking through Help items — we already found it
                last;
            }
        }

        last;  # Stop looking through top-level menus — Help is unique
    }

    # ------------------------------------------------
    # Step 5: Reassign the menuItems structure
    # This tells Prima to redraw the menu so the label change appears
    # ------------------------------------------------
    $mw->menuItems($menu_items);
}
```

Listing 22.3 Dynamic Menu Modifying a Label

In a File menu, MRU stands for Most Recently Used. It's a list of the files that were opened or edited most recently in the application. This feature helps users quickly reopen recent files without needing to browse for them again. An example is the 'Recent Files' entry in my preferred lightweight GUI text editor, Geany.

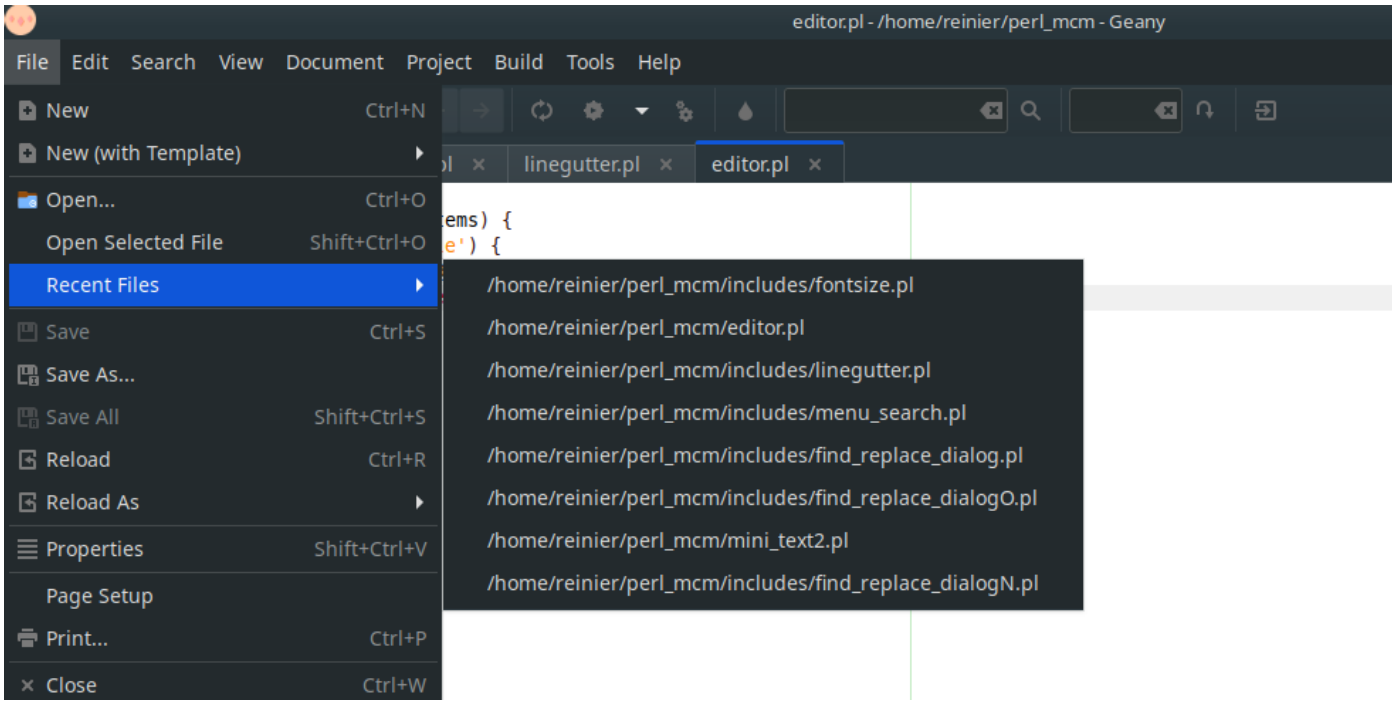


Figure 22.4: Submenu MRU in Geany

The key idea is to replace only the submenu array, not the whole menu. There is more than one solution. Here I only provide the fastest, while avoiding redundant parsing and memory operations. To give you an initial idea, the following code replaces the previous MRU list with a new one.

```
use Prima qw(Application Menus);

my $mw;
my $menu_items;
my $mru_menu;

sub update_recent_files {
    foreach my $menu (@$menu_items) {
        next unless $menu->[0] eq 'File';
        # Look inside the "File" menu items
        foreach my $item (@{$menu->[1]}) {
            # Find the item with ID2 (the Recent files submenu)
            if (ref($item) eq 'ARRAY' and $item->[0] eq 'ID2') {
                # Replace the submenu items with a new list
                $item->[2] = [
                    ['file3.txt' => sub { print "File 3\n"; }],
                    ['file4.txt' => sub { print "File 4\n"; }],
                ];
                last;
            }
        }
        last;
    }

    # Reassign only the menuItems — keeps same menu object
    $mw->menuItems($menu_items);

    # print "Updated Recent files submenu!\n";
}

sub build {
    $menu_items = [
        ['File' => [
            ['ID1', '~Open', 'Ctrl+O', '^O', sub { update_recent_files(); }],
            # "Recent files" submenu (initial entries from $mru_menu)
            ['ID2', 'Recent files' => $mru_menu ],
            [],
            ['ID3', '~Exit', 'Alt+X', km::Alt | ord('x'), sub { shift->close }],
        ]],
    ];
    return $menu_items;
}

# ------------------------------------------------------------
# Initial contents of the "Recent files" submenu.
# These will later be replaced by update_recent_files().
# ------------------------------------------------------------
$mru_menu = [
    ['file1.txt' => sub { print "File 1\n"; }],
    ['file2.txt' => sub { print "File 2\n"; }],
];

$mw = Prima::MainWindow->new(
    text => 'Efficient Menu Update',
    menuItems => build(),
);

Prima->run;
```

Listing 22.4: Fastest MRU Implementation (with In-place Modification)

For a Geany-like MRU list, you need a file that has a MRU list, such as my mru.txt, consisting of files I opened recently :

```
/home/reinier/perl_mcm/data/repetition.mcm
/home/reinier/perl_mcm/data/bach_bwv565.abc
/home/reinier/bmt/schumann_WO024.abc
...
```

Load this file when the program starts to (re)build the MRU menu.

```
sub load_mru {
    my $file = 'mru.txt';

    return [] unless -e $file;   # no MRU file yet

    open my $fh, '<', $file or return [];

    my @MRU_items;
    my $i = 0;
    while (my $line = <$fh>) {
        chomp $line;
        next if $line eq '';      # skip empty lines

        $i++;
        # defining the MRU menu items
        push @MRU_items, [
            '',                    # empty ID
            "$i: $line",           # number + filename as single string
            sub { open_mru_file($line) } # callback
        ];
    }

    close $fh;

    return \@MRU_items;
}
```

Listing 22.5: Reading mru.txt and Return \@MRU_items

Use this as follows:

```
['ID2', 'Recent files' => load_mru()],
```

So what happens if you open a new file or save your text into a new file? Well, basically this:

1. Update mru.txt with the current file at the top
2. Remove any duplicates

```
my $mru_file = "mru.txt";
my $new_entry = "/home/reinier/bmt/schumann_WO024.abc";   # reopen a file that exist in mru.txt

# Read existing MRU entries
open my $in, "<", $mru_file or die "Cannot open $mru_file: $!";
my @lines = <$in>;
close $in;

chomp @lines;

# Remove all occurrences of the new entry
@lines = grep { $_ ne $new_entry } @lines;

# Add new entry at the top
unshift @lines, $new_entry;

# Write new list back to file
open my $out, ">", $mru_file or die "Cannot write $mru_file: $!";
print $out "$_\n" for @lines;
close $out;
```

Listing 22.6: Updating mru.txt

Invoke now the code of Listing 22.5 and your MRU-list is updated.

23. A Full Application: Building a Text editor

The following sections demonstrate how menus integrate into a real application. Each step adds new functionality: building the editor, adding preferences, file operations, and search/replace.

23.1 Base Structure of the Editor

We begin with a simple editor application containing:

- a menu bar (File, Edit, Help), and
- a central `Prima::Edit` widget.

This establishes the foundation upon which later features - preferences, file dialogs, and search/replace - are built.

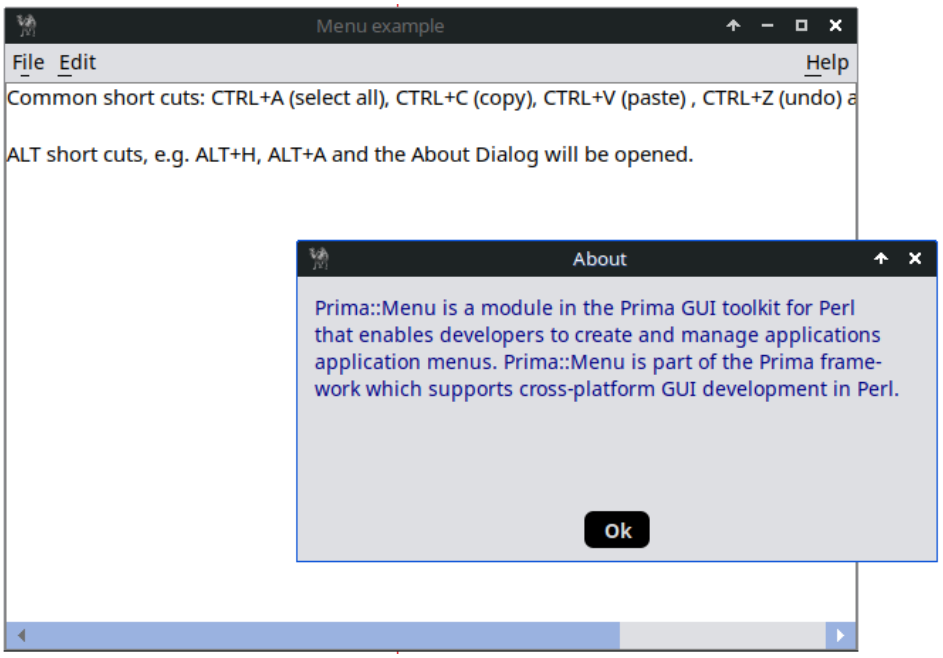


Figure 23.1: Editor with Basic Menu

```
use Prima qw( Label Edit Buttons ComboBox Application );
use Prima::Menus;

use lib '.';
# my custom message dialog
require "showMessageDialog.pl";

my $editor;
my $mw;

$mw = Prima::MainWindow->new(
    text      => "Menu example",
    size      => [600, 400],
    backColor => cl::White,
    icon  => Prima::Icon->load('icon.png'),

   menuItems =>
   [
    [ 'F~ile' => [
                    [ 'New', 'Ctrl+N', '^N', sub {

                                $editor->set(text => '');
                                $editor->insert_text("You clicked NEW")

                               }
                                                                  ],
                     [], # divisor line
                     [ 'Exit', 'Ctrl+Q', '^Q', sub { exit } ],
                ]
    ],

    [ '~Edit' => [
                    ['Undo' => 'Ctrl+Z' => '^Z', sub { $editor->undo }],
                                ['Redo' => 'Ctrl+R' => '^R' , sub { $editor->redo }],
                                [],
                                ['Cut' => 'Ctrl+X' => '^X', sub{ $editor->cut }],
                                ['Copy' => 'Ctrl+C' => '^C', sub{ $editor->copy }],
                                ['Paste' => 'Ctrl+V' => '^V', sub{ $editor->paste }],
                                [],
                                ['Delete' => 'Del' => kb::NoKey, sub{
                                                $editor->delete_block }
                    ],
                                ['Select all' => 'Ctrl+A' => '^A', sub{
                                                $editor->select_all }
                    ],
                         ]
    ],

    [], # divisor in main menu: menuitem Help are now at the right

    [ '~Help' => [

        ['~About' => sub{ showMessageDialog(450, 200, "About",
            "Prima::Menu is a module in the Prima GUI toolkit for Perl\n" .
            "that enables developers to create and manage applications\n" .
                    "application menus. Prima::Menu is part of the Prima frame-\n" .
            "work which supports cross-platform GUI development in Perl.",
                                                            ta::Left);}],
        ]
     ],
    ],
```

```
);


$editor = $mw->insert( Edit =>

    pack => { fill => 'both', expand => 1, side => 'left', pad => 0 },
    size => [500, 300],
    text => "Common short cuts: CTRL+A (select all), CTRL+C (copy), " .
            "CTRL+V (paste) , CTRL+Z (undo) and Delete work\n\nALT " .
            "short cuts, e.g. ALT+H, ALT+A and the About Dialog " .
            "will be opened.",
    font => { size => 11, },
    color => cl::Black,
    backColor => 0xFFFFFF,

    autoSelect => 0,

    growMode => gm::Client,
    name => 'Editor',
    popupItems  => [

         ['Undo' => 'Ctrl+Z' => '^Z', sub{ $_[0]->undo }],
         ['Redo' => 'Ctrl+Y' => '^Y', sub{ $_[0]->redo }],
         [],
         ['Cut'        => 'Ctrl+X'    => '^X', sub{ $_[0]->cut }],
         ['Copy'       => 'Ctrl+C'    => '^C', sub{ $_[0]->copy }],
         ['Paste'      => 'Ctrl+V'    => '^V', sub{ $_[0]->paste }],
         [],
         ['Delete'     => 'Del'       => kb::NoKey, sub{ $_[0]->delete }],
         ['Select all' => 'Ctrl+A'    => '^A' => sub { $_[0]->select_all }],

    ],

);

$editor->set( current => 1 );
$editor->cursor_cend();

Prima->run;
```

Listing 23.1: Editor with Basic Menu

In the following snippets, only new or modified code will be shown.

23.2 Preferences

The menu bar is extended with a "Preferences" menu item, which contains three submenu options.

- The "Font Size" option loads an external file ‘fontsize.pl’ with a simple list box for selection (see below).
- The "Word Wrap" option is a checkbox that toggles between on and off by setting the `wordWrap` property.
- The "Background Color" is set by assigning a color value to the `backColor` property.
- The "Background Color Header" is a disabled submenu item.

These options demonstrate the use of special characters such as `*`, `@`, and `-`.

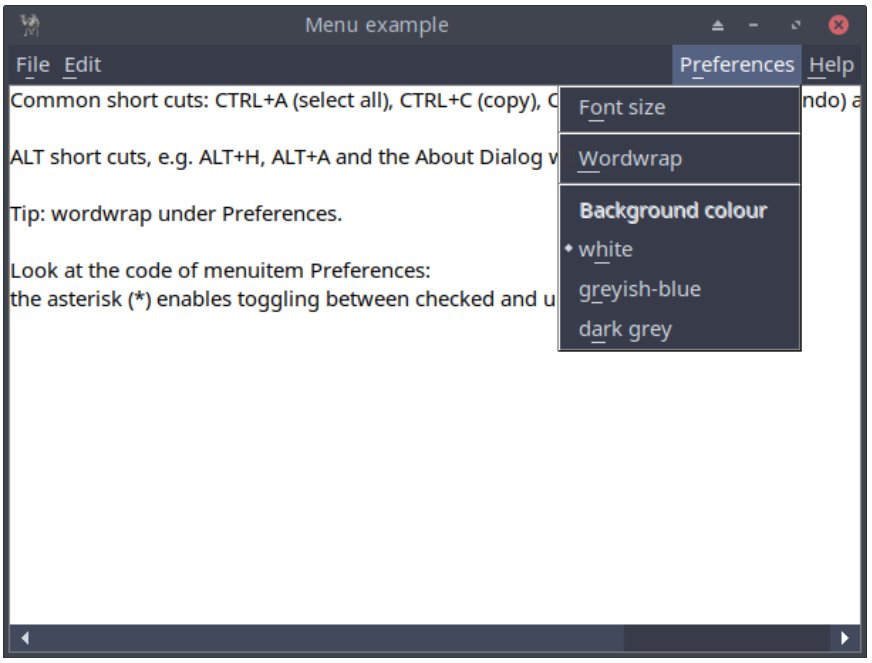


Figure 23.2: Basic Menu Extended with Preferences Menu Item

```
require "fontsize.pl";

my $editor;
my $mw;

$mw = Prima::MainWindow->new(
    (...)

    [], # divisor in main menu: menuitem Preferences and Help are now at the right

    [ 'P~references' => [
        [ 'Font size', 'F~ont size', '', '',
                sub {
                     my $retVal = mcFontsize();
                     $editor->set( font => { size => $retVal } );}
    ],
    [],
    [ "@" =>  "~Wordwrap" => sub { ($_[2]) ? $editor->set( wordWrap => 1 ) :
                                        $editor->set( wordWrap => 0 ); } ],
    [],
    [ '-','Background colour', '', ],
    [ '*(' => 'w~hite' => sub { $editor->backColor(cl::White), } ],
    [ '' => 'g~reyish-blue' => sub { $editor->backColor(0x839496), } ],
    [ ')' => 'd~ark grey' => sub { $editor->backColor(0x4F4F4F), } ],

       ]
    ],


    (...)

);
```

Listing 23.2: Basic Menu Extended with Preferences Menu Item

And *fontsize.pl*:

```
# fontsize.pl

# default value
my $retVal = 11;
sub mcFontsize {

     my $popup = Prima::Dialog->create(
            size => [300,200],
            text => "Select a font size",
            centered => 1,
            icon => Prima::Icon->load('icon.png'),
     );

     $lb = $popup->insert( "ListBox",
      name           => 'ListBox1',
      origin         => [50, 80],
      size           => [200, 100],
      multiSelect    => 0, # if 0, the user can select only one item.code_comment>
      font           => { size => 10},
      items          => [ '8', '9', '10', '11', '12', '14' ],
      focusedItem    => 3,
      align          => ta::Left,

     );

      $popup->insert(
            Button =>
                pack => { fill => 'none', side => 'bottom', pad => 15 },
                size    => [50, 30],
                text    => "Ok",
                onClick => sub {

                        my $mcSelectedOptions = "";

                        foreach (@{ $lb->selectedItems }){
                             $mcSelectedOptions = $lb->items->[$_];
                        };

                        $retVal = $mcSelectedOptions;
                        $popup->destroy;
                }
           );

       # execute brings the widget in a modal state
           $popup->execute();
           return($retVal);
}

# return a true value to the interpreter
1;
```

Listing 23.3: File fontsize.pl

23.3 File Options

We add now three File menu items that invoke built-in dialogs: Open, Save, and Save As. First, we need to extend the *use Prima* line to include *Dialog::FileDialog*:

```
        use Prima qw(Label Edit Buttons ComboBox Application Dialog::FileDialog);
```

Prima::Dialog::FileDialog provides a standard file dialog that allows users to navigate the file system and select one or multiple files. The class supports two modes: open and save, which are internally triggered by *Prima::Dialog::OpenDialog* and *Prima::Dialog::SaveDialog*, respectively.

The menu items Open, Save, and Save As are connected to functions with predictable names:

```
[ 'Open', 'Ctrl+O', '^O', sub { open_file_dialog  } ],
[],
[ 'Save', 'Ctrl+S', '^S', sub { save_file_dialog  } ],
[ 'Save as', '', '', sub { save_as_file_dialog  } ],
```

The method *open_file_dialog* invokes *Prima::Dialog::OpenDialog*, using a filter option

```
my $open = Prima::Dialog::OpenDialog->new(
        filter => [ ['TXT files' => '*.txt'],  ['All' => '*'] ]
);
```

The method *save_as_file_dialog* invokes *Prima::Dialog::SaveDialog*.

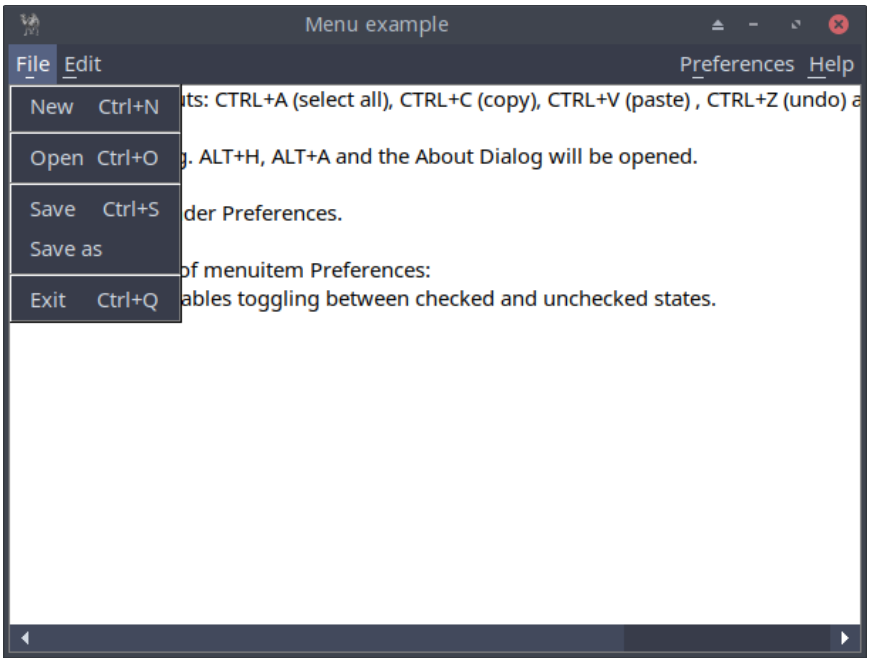


Figure 23.3: Extended File Menu Item

```
[ 'F~ile' => [
       [ 'New', 'Ctrl+N', '^N', sub { $current_filename = '';
                                      $editor->set(text => '');
                                      $editor->insert_text("You clicked NEW")

                                     }
                                                                    ],
        [], # divisor line
        [ 'Open', 'Ctrl+O', '^O', sub { open_file_dialog  } ],
        [],
        [ 'Save', 'Ctrl+S', '^S', sub { save_file_dialog  } ],
        [ 'Save as', '', '', sub { save_as_file_dialog  } ],
         [],
         [ 'Exit', 'Ctrl+Q', '^Q', sub { exit } ],
                                ]
],

##############################################################################
# open_file_dialog
##############################################################################

my $current_filename;

sub open_file_dialog {

    my $open = Prima::Dialog::OpenDialog->new(
        filter => [ ['TXT files' => '*.txt'],  ['All' => '*'] ]
    );

    if ($open->execute) {

        open(my $fh, '<', $open->fileName) or
                   die "Cannot open file '$open->fileName' for reading: $!";

        local $/;
        my $file_content = <$fh>;
        $editor->set( text => $file_content );
        $mw->set( text => $open->fileName );

        close($fh);

        $current_filename = $open->fileName;
    }
}

##############################################################################
# save_as_file_dialog
##############################################################################

sub save_as_file_dialog {
    my $save = Prima::Dialog::SaveDialog->new(
    );

    if ($save->execute) {

        my $content = $editor->text;

        $current_filename = $save->fileName;

        open(my $fh, '>', $save->fileName) or
                   die "Cannot open file '$save->fileName' for writing: $!";
```

```
        print $fh $content;
        $mw->set( text => $save->fileName );
        close($fh);
    }
}

################################################################################
# save_file_dialog
################################################################################
sub save_file_dialog {

        my $content = $editor->text;

        if (defined($current_filename) ne "") {

            open(my $fh, '>', $current_filename) or
                 die "Cannot open file '$current_filename' for writing: $!";
            print $fh $content;
            close($fh);
        }
        else {
                    save_as_file_dialog();
        }
}
```

Listing 23.4: Extended File Menu Item

23.4 Search/Replace Options: the hard way

The search and replace functions use the built-in dialogs `Prima::Dialog::FindDialog`, as well as the standard `FindDialog` and `ReplaceDialog`, to locate and replace text. Their implementation, however, can be somewhat challenging. But why reinvent the wheel? The *examples* directory of the Prima package includes an `editor.pl` script, which is fairly complex.

For this reason, I created a simpler and more compact version - an application containing only the Edit menu. I will present this version in a few steps. After going through them, you should be able to understand the full Prima example more easily.

23.4.1 `profile_default` Method

This method defines default properties for the window, including menu items.

```
sub profile_default {
    my %def = %{$_[0]->SUPER::profile_default};
    return {
        %def,
        fileName => undef,
        menuItems => [...],
    };
}
```

`my %def = %{$_[0]->SUPER::profile_default};` fetches the default profile (e.g., window size, title, etc.) from the parent class (`Prima::Window`). It creates a new hash (`%def`) and copies all key-value pairs from the parent's default profile into it (%{...} dereferences the hash reference returned by `SUPER::profile_default`). The child class (`EditorWindow`) can then add or override these defaults without modifying the parent's original hash.

`$_[0]`: the first argument passed to the method (in this case, the object or class invoking `profile_default`.

23.4.2 `init` Method

This method initializes the editor window.

```
sub init {
    my ($self, %profile) = @_;
    %profile = $self->SUPER::init(%profile);
    my $fn = $profile{fileName} || '.Untitled';
    $self->{editor} = $self->insert(
        'Prima::Edit' =>
        name => 'Edit',
        # binds the editor's text to a scalar reference (`$cap`)
        textRef => \my $cap,
        origin => [0, 0], size => [$self->width, $self->height],
        hScroll => 1,
        vScroll => 1,
        growMode => gm::Client, # ensures the editor resizes with the window
        wordWrap => 1,
    );
    $self->text($fn);
    $self->{editor}->focus;
    # initializes a placeholder for find/replace data
    $self->{findData} = undef;
    return %profile;
}
```

23.4.3 `find_dialog` Method

```
sub find_dialog {
    # $findStyle: if 1, it's a Find operation; if 0, it's Replace.
    my ($self, $findStyle) = @_;
    %{$self->{findData}} = (
        replaceText => '', findText => '', replaceItems => [],
                           findItems => [],
                           options => 0, scope => fds::Cursor,
    ) unless $self->{findData};
    my $fd = $self->{findData};
    # %prf: pre-fills the dialog with previous search/replace data
    my %prf = map { $_ => $fd->{$_} } qw(findText options scope);
    $prf{replaceText} = $fd->{replaceText} unless $findStyle;
    $findDialog ||= Prima::Dialog::FindDialog->new;
    $findDialog->set(%prf, findStyle => $findStyle);
    $findDialog->Find->items($fd->{findItems});
    $findDialog->Replace->items($fd->{replaceItems}) unless $findStyle;
    my $rf = $findDialog->execute;
    if ($rf != mb::Cancel) {
        for (qw(findText options scope)) {
            $self->{findData}{$_} = $findDialog->$_();
        }
        $self->{findData}{replaceText} = $findDialog->replaceText()
                                         unless $findStyle;
        $self->{findData}{result} = $rf;
        $self->{findData}{asFind} = $findStyle;
        @{$self->{findData}{findItems}} = @{$findDialog->Find->items};
        @{$self->{findData}{replaceItems}} = @{$findDialog->Replace->items}
                                             unless $findStyle;
        return 1;
    }
    return 0;
}
```

23.4.4 `do_find` Method (The Core Logic)

```
sub do_find {
    my $self = $_[0];
    my $e = $self->{editor};
    my $p = $self->{findData};
    my (@start, @end);
    my $success;
    my @sel = $e->has_selection ? $e->selection : ();

    FIND: {
        # @start and @end Logic:
        # - determines where to start/end the search based on scope and
        #   current selection
        # - uses ternary operators to handle different cases
        # (e.g., fds::Top, fds::Bottom)

        # Calculate @start
        if ($$p{scope} != fds::Cursor) {
            if (@sel) {
                @start = $$p{scope} == fds::Top ? ($sel[0], $sel[1])
                                                : ($sel[2], $sel[3]);
            } else {
                @start = $$p{scope} == fds::Top ? (0, 0) : (-1, -1);
            }
        } else {
            @start = $e->cursor;
        }

        # Calculate @end
        if (@sel) {
            @end = $$p{scope} == fds::Bottom ? ($sel[0], $sel[1])
                                             : ($sel[2], $sel[3]);
        } else {
            @end = $$p{scope} == fds::Bottom ? (0, 0) : (-1, -1);
        }

        # Perform the search
        # $e->find() returns the coordinates of the found text (@n).
        my @n = $e->find($$p{findText}, @start, $$p{replaceText},
                                                $$p{options}, @end);
        unless (defined $n[0]) {
            message_box($self->text, $success ? ("All done", mb::Information)
                                              : ("No matches found",
                                              mb::Error), compact => 1);
            return;
        }

        # Highlight the found text
        my $nx = $n[0] + length($p->{replaceText});
        $e->cursor($$p{options} & fdo::BackwardSearch ? $n[0] : $n[2], $n[1]);
        # physical_to_visual: converts physical coordinates (used
        # internally by Prima) to visual coordinates (for display).
        $_ = $e->physical_to_visual($_, $n[1]) for @n[0, 2];
        # $e->selection: highlights the match
        $e->selection($n[0], $n[1], $n[2], $n[1]);

        # Handle Replace logic
        unless ($$p{asFind}) {
            if ($$p{options} & fdo::ReplacePrompt) {
                my $r = message_box($self->text,
                                    "Replace this text?", mb::YesNoCancel |
                                    mb::Information | mb::NoSound, compact => 1);
```

```
                redo FIND if $r == mb::No && $$p{result} == mb::ChangeAll;
                last FIND if $r == mb::Cancel;
            }
            # $e->set_line: updates the text if replacing
            $e->set_line($n[1], $n[3]);
            $e->cursorX($nx) unless $$p{options} & fdo::BackwardSearch;
            $success = 1;
            redo FIND if $$p{result} == mb::ChangeAll;
        }
    }
}
```

23.4.5 Helper Methods

Three wrapper methods (Find, Replace, Find Next), activated via menu items.

```
sub find    { my $s = $_[0]; return unless $s->find_dialog(1);  $s->do_find; }
sub replace { my $s = $_[0]; return unless $s->find_dialog(0);  $s->do_find; }
# reuses the last search term (`$s->{findData}`) and calls `do_find`
sub find_next { my $s = $_[0]; return unless $s->{findData}; $s->do_find; }
```

23.4.6 Result

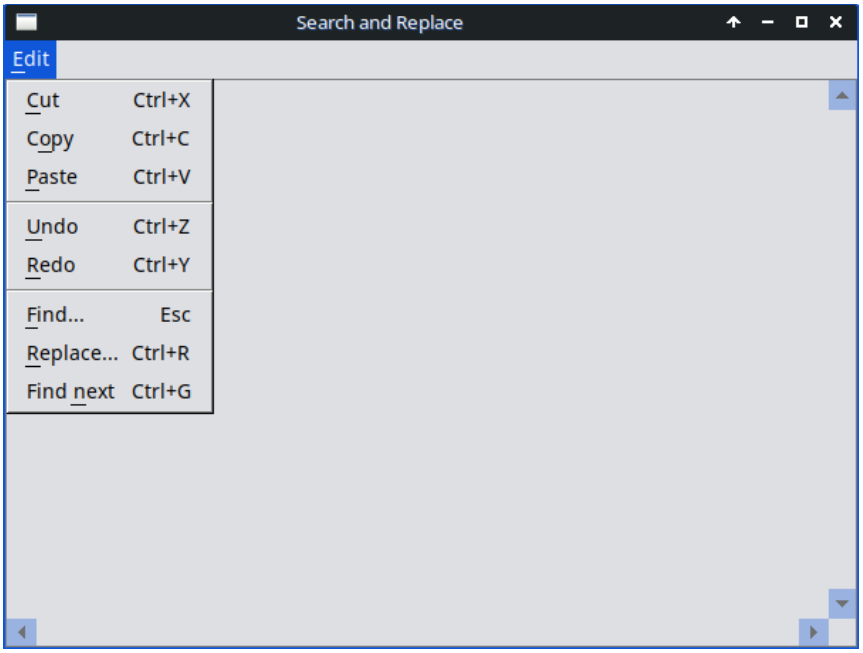


Figure 23.4: Edit Menu

Complete code:

```
use Prima qw(Edit Application MsgBox Dialog::FindDialog);

package EditorWindow;
use base qw(Prima::Window);

sub profile_default {
    my %def = %{$_[0]->SUPER::profile_default};
    return {
        %def,
        fileName => undef,
        menuItems => [
            ['~Edit' => [
                ['~Cut'      => 'Ctrl+X' => kb::NoKey,
                                sub { $_[0]->{editor}->cut }],
                ['C~opy'     => 'Ctrl+C' => kb::NoKey,
                                sub { $_[0]->{editor}->copy }],
                ['~Paste'    => 'Ctrl+V' => kb::NoKey,
                                sub { $_[0]->{editor}->paste }],
                [],
                ['~Undo'     => 'Ctrl+Z' => kb::NoKey,
                                sub { $_[0]->{editor}->undo }],
                ['~Redo'     => 'Ctrl+Y' => kb::NoKey,
                                sub { $_[0]->{editor}->redo }],
                [],
                ['~Find...'    => 'Esc'     => kb::Esc, q(find)],
                ['~Replace...' => 'Ctrl+R'  => '^R',    q(replace)],
                ['Find ~next'  => 'Ctrl+G'  => '^G',    q(find_next)],
            ]],
        ],
    };
}

sub init {
    my ($self, %profile) = @_;
    %profile = $self->SUPER::init(%profile);
    my $fn = $profile{fileName} || '.Untitled';
    $self->{editor} = $self->insert(
        'Prima::Edit' => name => 'Edit', textRef => \my $cap,
        origin => [0, 0], size => [$self->width, $self->height],
        hScroll => 1, vScroll => 1, growMode => gm::Client, wordWrap => 1,
    );
    $self->text($fn);
    $self->{editor}->focus;
    $self->{findData} = undef;
    return %profile;
}

sub on_destroy { $::application->close }

# --- Find/Replace Logic (unchanged) ---
my $findDialog;

sub find_dialog {
    my ($self, $findStyle) = @_;
    %{$self->{findData}} = (
        replaceText => '', findText => '', replaceItems => [],
                           findItems => [],
                           options => 0, scope => fds::Cursor,
    ) unless $self->{findData};
    my $fd = $self->{findData};
    my %prf = map { $_ => $fd->{$_} } qw(findText options scope);
    $prf{replaceText} = $fd->{replaceText} unless $findStyle;
```

```
    $prf{replaceText} = $fd->{replaceText} unless $findStyle;
    $findDialog ||= Prima::Dialog::FindDialog->new;
    $findDialog->set(%prf, findStyle => $findStyle);
    $findDialog->Find->items($fd->{findItems});
    $findDialog->Replace->items($fd->{replaceItems}) unless $findStyle;
    my $rf = $findDialog->execute;
    if ($rf != mb::Cancel) {
        for (qw(findText options scope)) {
            $self->{findData}{$_} = $findDialog->$_();
        }
        $self->{findData}{replaceText} = $findDialog->replaceText() unless $findStyle;
        $self->{findData}{result} = $rf;
        $self->{findData}{asFind} = $findStyle;
        @{$self->{findData}{findItems}} = @{$findDialog->Find->items};
        @{$self->{findData}{replaceItems}} = @{$findDialog->Replace->items}
                                            unless $findStyle;
        return 1;
    }
    return 0;
}

sub do_find {
    my $self = $_[0];
    my $e = $self->{editor};
    my $p = $self->{findData};
    my (@start, @end);
    my $success;
    my @sel = $e->has_selection ? $e->selection : ();
    FIND: {
        # Calculate @start
        if ($$p{scope} != fds::Cursor) {
            if (@sel) {
                @start = $$p{scope} == fds::Top ? ($sel[0], $sel[1])
                                                : ($sel[2], $sel[3]);
            } else {
                @start = $$p{scope} == fds::Top ? (0, 0) : (-1, -1);
            }
        } else {
            @start = $e->cursor;
        }

        # Calculate @end
        if (@sel) {
            @end = $$p{scope} == fds::Bottom ? ($sel[0], $sel[1])
                                             : ($sel[2], $sel[3]);
        } else {
            @end = $$p{scope} == fds::Bottom ? (0, 0) : (-1, -1);
        }

        my @n = $e->find($$p{findText}, @start, $$p{replaceText}, $$p{options}, @end);
        unless (defined $n[0]) {
            message_box($self->text, $success ? ("All done", mb::Information)
                                              : ("No matches found", mb::Error),
                                                 compact => 1);
            return;
        }
        my $nx = $n[0] + length($p->{replaceText});
        $e->cursor($$p{options} & fdo::BackwardSearch ? $n[0] : $n[2], $n[1]);
        $_ = $e->physical_to_visual($_, $n[1]) for @n[0, 2];
        $e->selection($n[0], $n[1], $n[2], $n[1]);
        unless ($$p{asFind}) {
            if ($$p{options} & fdo::ReplacePrompt) {
                my $r = message_box($self->text,
```

```
                                    "Replace this text?", mb::YesNoCancel |
                                    mb::Information | mb::NoSound, compact => 1);
                redo FIND if $r == mb::No && $$p{result} == mb::ChangeAll;
                last FIND if $r == mb::Cancel;
            }
            $e->set_line($n[1], $n[3]);
            $e->cursorX($nx) unless $$p{options} & fdo::BackwardSearch;
            $success = 1;
            redo FIND if $$p{result} == mb::ChangeAll;
        }
    }
}

sub find    { my $s = $_[0]; return unless $s->find_dialog(1);  $s->do_find; }
sub replace { my $s = $_[0]; return unless $s->find_dialog(0);  $s->do_find; }
sub find_next { my $s = $_[0]; return unless $s->{findData}; $s->do_find; }

EditorWindow->new(
    origin => [10, 100],
    size   => [600, 400],
    fileName => "Search and Replace",
    font => { size => 14, name => 'Courier New' },
);
Prima->run;
```

Listing 23.5: Find and Replace (Adapted Code form editor.pl Prima Package)

23.4.7 Study Tip: Understanding the *editor.pl* Example

Now that you've explored the code, you should be able to fully understand the *editor.pl* example provided in the Prima toolkit package. Pay close attention to the integration of the status bar - this component plays a key role in improving user feedback and interface usability in the editor window (*package Indicator*).

23.5 Search/Replace Options: the easy variant

The built-in search and replace dialogs work well, but they can still be a bit cumbersome. To be honest, I often struggle with the dialog: I forget to set the Scope option and end up with no results. I also find it a bit frustrating that the dialog closes every time I run a search or replace. And while the code is excellent - I wish I were that good a programmer - it still feels a little complex.

So I asked myself: *Can I make the code easier to understand and replace the built-in dialog with a custom one?* Maybe even split it into two separate dialogs - a Find dialog and a Replace dialog.

It's a nice exercise to test whether I fully understand the original code, and while working on it, I realized there were a few things I had overlooked… Let's begin.

23.5.1 A Find and Replace dialog

I decided to make two separate dialogs with only a few options:

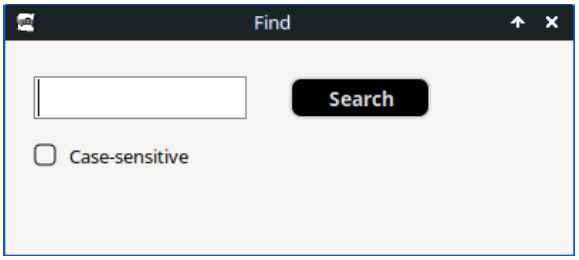


Figure 23.5: Custom Find Dialog with Only One Option

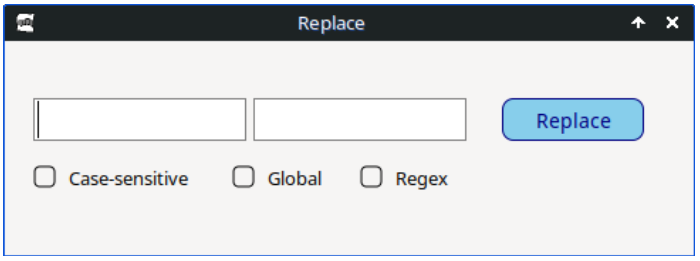


Figure 23.6: Custom Replace Dialog with Three Options

23.5.2 Find dialog

With a `sub { find_dialog($editor) }` the following code is invoked (and for displaying messages I use `require "includes/messageboxes.pl";`):

```
sub find_dialog {

        my ($editor) = @_;

        my $mw = Prima::Window->new(
                text => "Find",
                backColor   => 0xf6f5f4,
                color => cl::Black,
                size => [400, 150],
                borderStyle => bs::Dialog,
                borderIcons => bi::SystemMenu|bi::TitleBar,
                icon => Prima::Icon->load('img/mcm.png'),
        );

        # default settings for editor and all inputlines
        my %inputline_settings = (
                text      => "",
                font      => { size => 12, },
                color     => cl::Black,
                backColor => 0xFFFFFF,
        );

        # default settings for all checkboxes
        my %checkbox_settings = (
                font      => { size => 10, },
                color     => cl::Black,
                ownerBackColor => 1,
                hiliteBackColor => cl::LightGray,
                hiliteColor => cl::Red,
        );

        my $height_buttons = 30;

        my $input_f = $mw->insert( InputLine =>

                origin => [20, 95],
                size => [150, $height_buttons],
                %inputline_settings,
        );

        my $mcCheck_CSF = $mw->insert( CheckBox =>
                origin => [20, 55],
                text => "Case-sensitive",
                %checkbox_settings,
        );

        # at this point, the find code begins
        my @found = ();
        my $x = 0;
        my $y = 0;

        $mw->insert( Button  =>

                origin => [200, 95],
                size    => [100, $height_buttons],
                text    => "Search",
                color   => cl::Blue,
                backColor => 0x87CEEB,
                default => 1,
                onClick => sub {
```

```
my $search_term = $input_f->text;
$search_term =~ s/^\s+|\s+$//g; # Trim whitespace
if (!$search_term) {
       mcMessage(200, 100, "Warning!",
                           "Please enter a search term!", ta::Center);
       return;
}

# reset $x and $y if the search term has changed
if (!defined $last_search_term || $last_search_term ne $search_term) {
    $x = 0;
    $y = 0;
    # Update the last search term
    $last_search_term = $search_term;
}

# check if both input field has non-empty, non-whitespace text
if ( $search_term =~ /\S/ ) {

        # init/reset and search from the beginning
        ($x = 0, $y = 0) if (! $found[2]);

        # Starting position for search (current cursor position)
        my @start = ($x, $y);
        # End position (-1,-1) means 'search to end of document'
        my @end = (-1,-1);

        my $flags = $mcCheck_CSF->checked ? fdo::MatchCase : 0;
        # perform the search
        @found = $editor->find($search_term, @start, "",
                               $flags, @end);

        # if a match is found
        if ( defined( $found[0] ) ) {

             # Retrieve the line where the match was found
             # $found[1] is guaranteed to exist if
             # $found[0] exists
             my $line = $editor->get_line($found[1]);

             # Move the search start forward:
             # - If match not at end of line: continue on same line
             # - Otherwise: move to next line and reset column to 0
             ( $found[0] < length($line) )
                  ? ( $x = $found[0] + 1, $y = $found[1] )
                                          : ( $y = $found[1] + 1, $x = 0 );

                  # Convert physical search coordinates to visual
                  # screen coordinates.
                  # This ensures the cursor and selection highlight
                  # correctly even when
                  # parts of the document are folded or wrapped.
                  $editor->physical_to_visual($editor, $found[1]) for @found[0, 2];

                  # Move cursor to the matched location
                  $editor->cursorY($found[1]);
                  $editor->cursorX($found[2]);

                  # highlight the found match in the editor
                  $editor->selection( $found[0], $found[1], $found[2], $found[1] );
         }
         else {
```

```
                                if ( !defined $found[0]) {
                                   mcMessage(200, 100, "Warning!", "No matches found!", ta::Center);
                                                                        return;
                                }
                        }
                }
        },
    );
}

1;
```

Listing 23.6: Code Find Dialog

23.5.3 Replace dialog

With a *`sub { replace_dialog($editor) }`* the following code is invoked (and for displaying messages I use again *`require "includes/messageboxes.pl";`*).

```
# Store previous search/replace terms globally so the dialog remembers
# whether the user changed them between invocations
my ($prev_search_term, $prev_replace_term);

sub replace_dialog {

    my ($editor) = @_;

    my $mw = Prima::Window->new(
        text => "Replace",
        #backColor => cl::LightGray,
        backColor   => 0xf6f5f4,
        color => cl::Black,
        size => [485, 150],
        borderStyle => bs::Dialog,
        borderIcons => bi::SystemMenu|bi::TitleBar,
        icon => Prima::Icon->load('img/icon.png'),
    );

    # default settings for editor and all inputlines
    my %inputline_settings = (
        text      => "",
        font      => { size => 12, },
        color     => cl::Black,
        backColor => 0xFFFFFF,
    );

    # default settings for all checkboxes
    my %checkbox_settings = (
        font      => { size => 10, },
        color     => cl::Black,
        ownerBackColor => 1,
        hiliteBackColor => cl::LightGray,
        hiliteColor => cl::Red,
    );

    my $height_buttons = 30;
    my $y_inputlines = 80;

    my $input_r1 = $mw->insert( InputLine =>
        origin => [20, $y_inputlines],
        size   => [150, $height_buttons],
        %inputline_settings,
    );

    my $input_r2 = $mw->insert( InputLine =>
        origin => [175, $y_inputlines],
        size   => [150, $height_buttons],
        %inputline_settings,
    );

    # Vertical spacing from input lines
    my $distance_from_inputlines = 40;

    my $mcCheck_CS = $mw->insert( CheckBox =>
        origin => [20, ($y_inputlines - $distance_from_inputlines)],
        text => "Case-sensitive",
        %checkbox_settings,
    );

    my $mcCheck_GL = $mw->insert( CheckBox =>
```

```
    origin => [160, ($y_inputlines - $distance_from_inputlines)],
    text => "Global",
    %checkbox_settings,
);

my $mcCheck_RA = $mw->insert( CheckBox =>
    origin => [250, ($y_inputlines - $distance_from_inputlines)],
    text => "Regex",
    %checkbox_settings,
);

# Used to detect "did we already replace something?"
$success = 0;

$mw->insert( Button =>
    origin => [350, $y_inputlines],
    size    => [100, $height_buttons],
    text    => "Replace",
    color   => cl::Blue,
    backColor => 0x87CEEB,
    onClick => sub {

        my $search_term = $input_r1->text;
        $search_term =~ s/^\s+|\s+$//g; # Trim whitespace
        unless ($search_term) {
            mcMessage(200, 100, "Warning!",
                                "Please enter a search term!", ta::Center);
                                return
        }

        my $replace_term = $input_r2->text;
        $replace_term =~ s/^\s+|\s+$//g;
        unless ($replace_term) {
           mcMessage(200, 100, "Warning!",
                               "Please enter a replace term!", ta::Center);
                               return
        }

        # Check if search/replace terms have changed
        if (!defined($prev_search_term) ||
            !defined($prev_replace_term) ||
            $search_term ne $prev_search_term ||
            $replace_term ne $prev_replace_term) {

                    # Reset search state
                    $success = 0;
                    $editor->cursorY(0);
                    $editor->cursorX(0);
                    $editor->selection(0, 0, 0, 0);

                    # Update previous terms
                    $prev_search_term = $search_term;
                    $prev_replace_term = $replace_term;
        }

        # Check if both input fields have non-empty, non-whitespace text
        if ( $search_term =~ /\S/ and $replace_term =~ /\S/ ) {

            my $options = 0;  # Start with no options

            # Add options based on your flags
            $options |= fdo::MatchCase if ($mcCheck_CS->checked);
            $options |= fdo::RegularExpression if ($mcCheck_RA->checked);
```

```
                $options |= re::REGEXP_EXTENDED if ($mcCheck_RA->checked);

                FIND: {
                    # determine start and end positions for search
                    my @sel   = $editor->has_selection ? $editor->selection
                                                                    : ();
                    my @start = @sel ? ($sel[0], $sel[1]) : (0,0);
                    my @end   = @sel ? ($sel[2], $sel[3]) : (-1,-1);

                    @found = $editor->find($search_term, @start,
                                           $replace_term, $options, @end);
                    # if match was found
                    if ( defined( $found[0] ) ) {

                        # convert physical to visual coordinates
                        $editor->physical_to_visual($editor, $found[1])
                                 for @found[0, 2];

                        $editor->cursorY($found[1]);
                        $editor->cursorX($found[2]);

                        # highlight the found match in the editor
                        $editor->selection( $found[0], $found[1],
                                            $found[2], $found[1] );

                        # apply modified line text
                        $editor->set_line( $found[1], $found[3] );
                        $success = 1;

                        # if Global is checked: continue replacing automatically
                        redo FIND if $mcCheck_GL->checked;

                    }
                    else {
                            if ( !defined $found[0]) {
                                my $message = ($success != 0) ?
                                              "Done. No matches found anymore!"
                                              :
                                              "No matches found!";
                                mcMessage(325, 100, "Warning!", $message, ta::Center);
                                return;
                            }
                    }
                }
            }
        }
    );
}

1;
```

Listing 23.7: Code Replace Dialog

23.5.4 Not comfortable with this code?

I'm totally comfortable with it. You're not? No worries at all! And now you can learn from AI! Ask for a refactoring e.g. into a more object-oriented design and AI can offer several approaches, for example:

- Classic Perl OO (using `package` and `bless {}`)
- A dedicated Prima widget subclass (probably most 'Prima-like')
- A lightweight OO wrapper around the existing code

Closing words

In this part, you brought together many of the skills learned throughout the book and applied them to the structure of complete applications. You explored how menus shape the user experience, how dynamic items like recent-files lists keep an interface flexible, and how icons add clarity and recognition.

Most importantly, you built a full text editor step by step - connecting menus, preferences, file operations, and search tools into a coherent whole. This demonstrates how individual widgets and techniques combine into a real, functional application.

Part 10 - Learning from the Visual Builder

The Prima toolkit includes a built-in visual form designer that lets you create user interfaces, eliminating the need to write UI code by hand. It functions much like other 'drag-and-drop' builders: you can place buttons, text boxes, and other widgets onto your window with a click. Adding commands or event handlers to widgets is straightforward as well. You can save your designs to a file and later open them either in your application.

While I don't personally use the Visual Builder in my workflow, I explored it for the purpose of this guide and discovered a few things that you might find useful.

24.1 What Can It Teach You?

When you design a UI in Prima's Visual Builder and export it as a `.pl` file, the generated code isn't just executable - it also serves as a live tutorial. It shows you:

- How Prima widgets are instantiated
- How widget properties are configured
- How layout and parent-child relationships are managed
- What defaults are automatically assumed or omitted
- How callbacks and event bindings are connected

This makes the Visual Builder a helpful tool for both rapid prototyping and learning the structure of Prima-based GUIs.

24.2 Learning from the Object Inspector

When you open Visual Builder, you'll see three main windows:

- Main Panel
- Form Window (where your widgets appear)
- Object Inspector (this one is especially useful for learning)

The Object Inspector shows the settings and behaviors of whatever widget you select - like a button or a text box.

How to Use It:

- First, add a widget (like a button) to the Form Window.
- Then, look at the left side of the Inspector. It shows either a list of *properties* (like size or color) or *events* (click the button below the list to switch between viewing *properties* and *events*.).
- The right side shows the current value or code for the item you selected, e.g. `$self = $_[0]`.

This is a great way to explore and learn how different widgets work. Just pick something (like a button), then explore its properties and events.

The following screenshots will make this clearer.

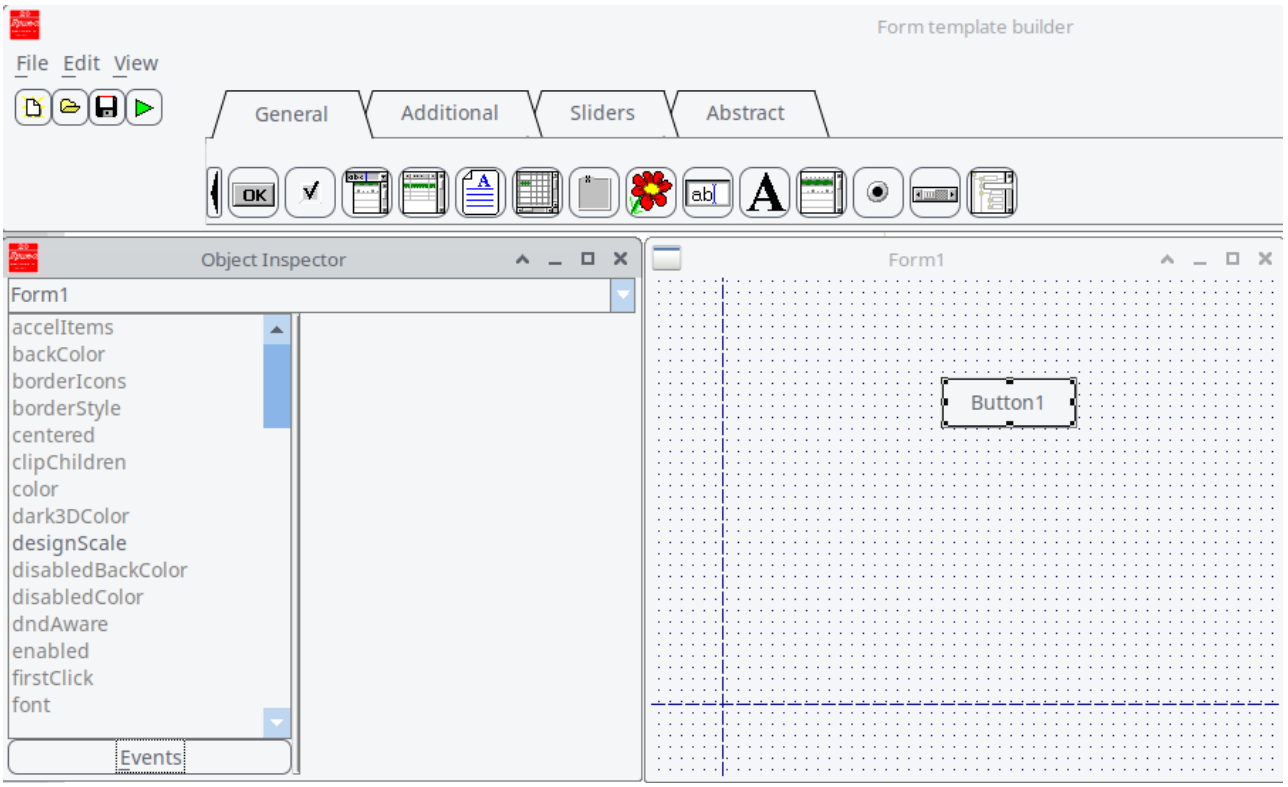


Figure 24.1: VB with Button Properties on the Left Side

Select property `backColor` and choose a color.

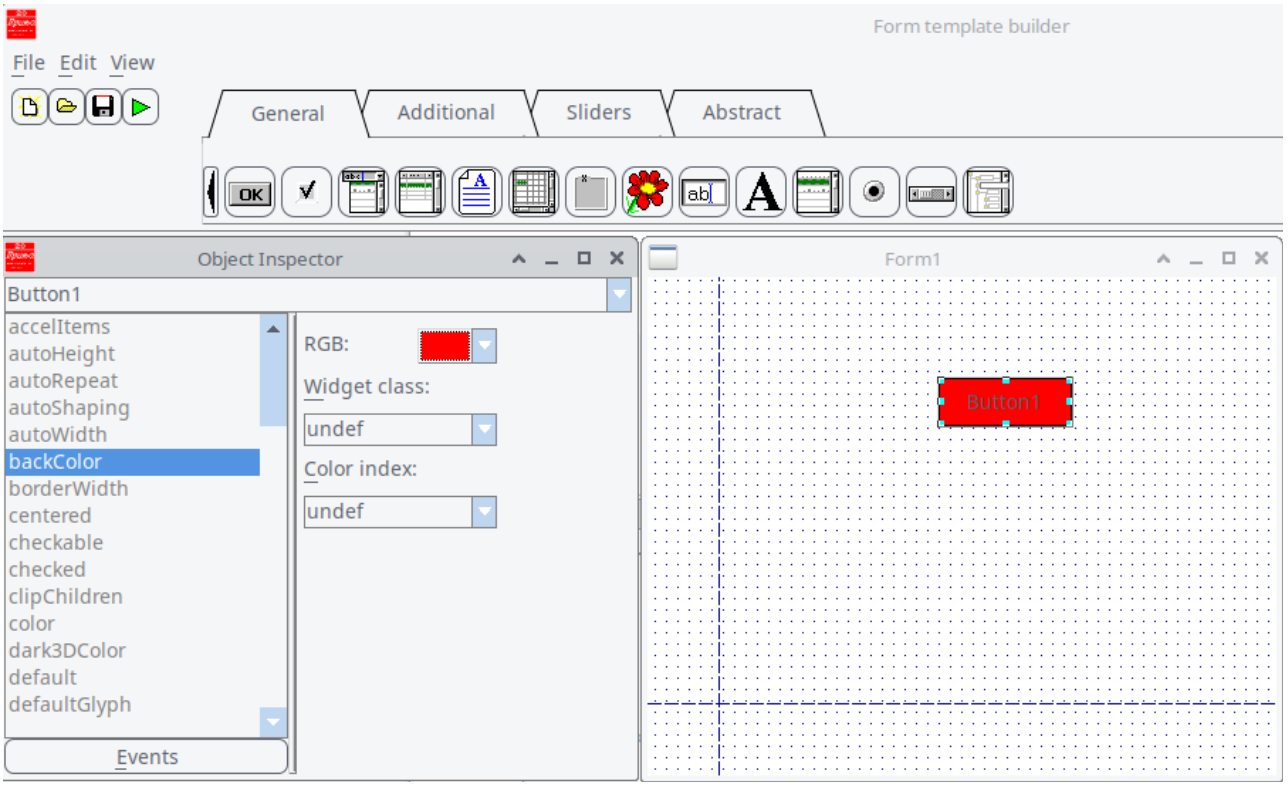


Figure 24.2: VB Using the Property backColor

Click Events below, you'll see a new list. Click the item `onClick`:

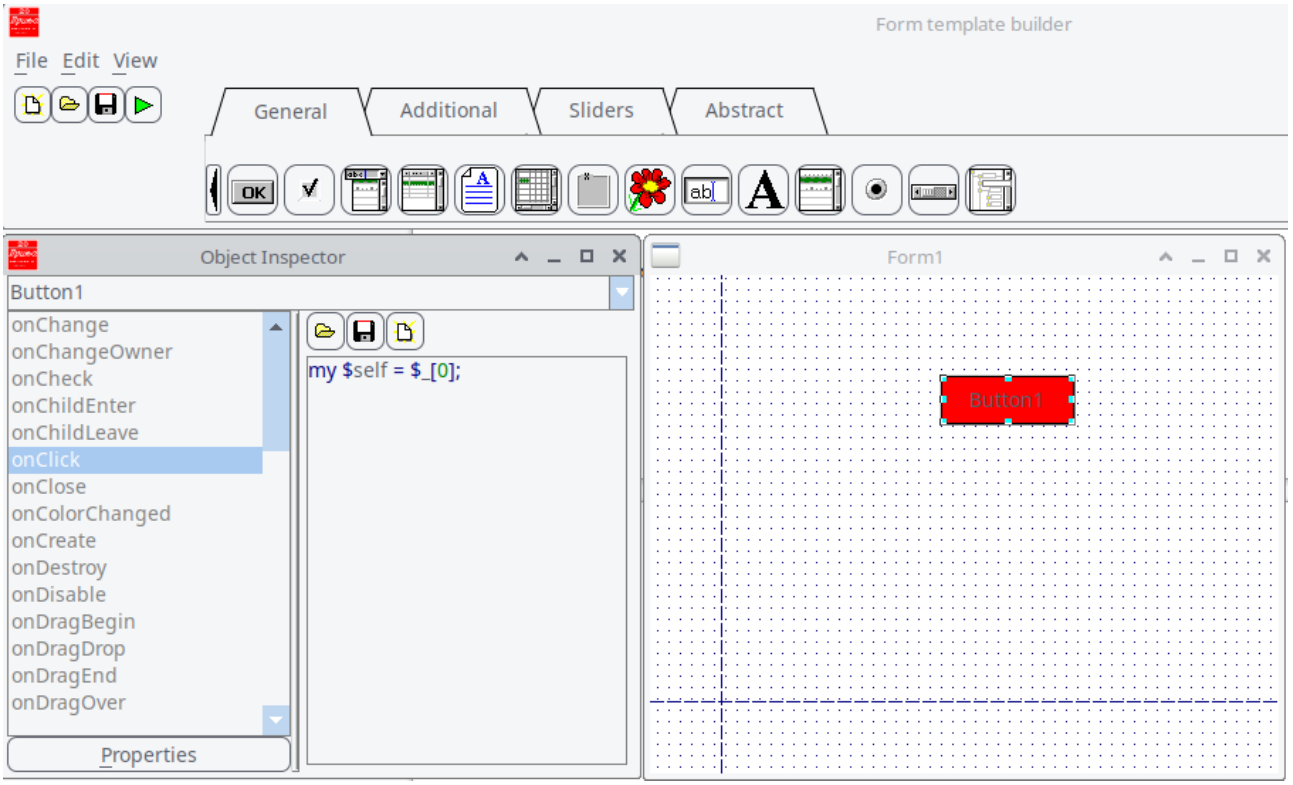


Figure 24.3: VB with Button Events on the Left Side

...and you learn the code my `$self = $_[0]`. Another example: click `onKeyDown` and you see the assignment of @_

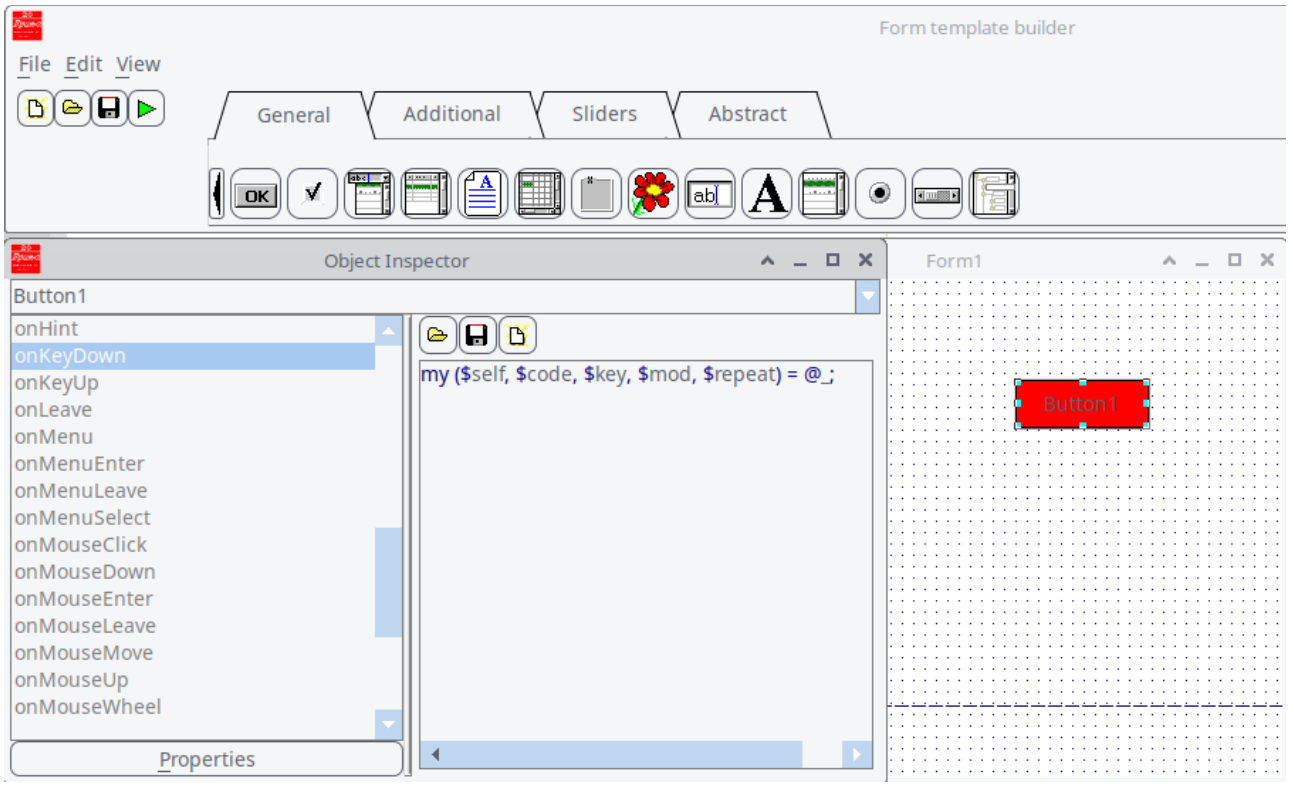


Figure 24.4: VB with Button Events on the Left Side; selecting onKeyDown Displays Information About the @_ array.

In this way, you can learn useful codes.

24.3 Learning from ‘Save as’

From the menu, choose File > Save As, and select Program scratch (*.pl) as the file type. This generates code that includes `profile-default` and `init` functions - common structures you’ll often see in Prima examples and modules.

Here’s how this program works, in plain terms:

- It creates a graphical window using the Prima toolkit for Perl.
- The window contains one red button.
- When you click the button, the background of the window turns green.

It follows a standard GUI lifecycle:

- Get default settings (`profile_default`)
- Build widgets (`init`)
- Show the window and run the app (`Prima->run`)

Below is the code, thoroughly annotated for clarity.

```
package Form1Window;
# Load the Prima GUI toolkit
use Prima;
use Prima::Classes;

# Declare this package as a subclass of Prima::MainWindow
use base 'Prima::MainWindow';

# Load support for Button widgets
use Prima::Buttons;

# This method provides default properties for the window.
# It is automatically called during ->new to get the initial setup values.
sub profile_default {
    # Get the default values from the superclass
    my $def = $_[0]->SUPER::profile_default;
    # Define our custom default properties for this window
    my %prf = (
        designScale => [8, 19],    # UI scaling factor (font, controls, etc.)
        name        => 'Form1',    # Internal name for the window widget
        origin      => [496, 268], # Starting position on screen (X, Y)
        size        => [607, 371], # Window dimensions (width x height)
    );

    # Merge our custom defaults into the inherited default hash
    @$def{keys %prf} = values %prf;
    # Return the merged default settings
    return $def;
}

# This method is called right after profile_default.
# It sets up widgets and layout inside the window.
sub init {
    my $self = shift;
    # Call the superclass's init method to complete base initialization
    my %profile = $self->SUPER::init(@_);
    # Prevent visual updates while adding widgets (for performance/smoothness)
    $self->lock;
    # Add a button to the window
    $self->insert('Prima::Button',
        name      => 'Button1',   # Internal widget name
        origin    => [105, 324],  # Position of button inside the window
        size      => [96, 36],    # Button size (width x height)
        backColor => 0xff0000,    # Background color (red, in hex)
        onClick   => sub {
            my $button = shift;
            # When the button is clicked, change the window background to green
            $button->owner->backColor(cl::Green);
        },
    );
    # Allow screen updates again
    $self->unlock;
    # Return the initialization profile (optional, but good practice)
    return %profile;
}

package main;
# Load the Prima application core (this sets up the app framework)
use Prima::Application;
# Create the main window. This calls:
#   1. profile_default to get default values
```

```
#   2. init to build the interface (e.g., button)
Form1Window->new;

Prima->run;
```

Listing 24.1: Package Form1Window and Package Main

However, the first click leaves white artifacts around the button after background color change. So I adapted the code.

```
onClick   => sub {
    my $button = shift;
    my $owner  = $button->owner;

    # Set new background color
    $owner->backColor(cl::Green);

    # Hide button to allow repaint under it
    $button->visible(0);

    # Force window redraw (including under the button)
    $owner->repaint;

    # Show the button again
    $button->visible(1);
},
```

and

```
# Paint background
sub onPaint {
    my ($self, $canvas) = @_;
    $canvas->color($self->backColor);
    $canvas->bar(0, 0, $self->size);
}
```

The trick was forcing the button to hide temporarily so the background beneath it could repaint properly.

24.4 A little modification

Just for fun, I modified the code of 24.3 and added a right-click option. So:

- Left-click: changes background to green
- Right-click: resets background to white

If you want to try, add to *`init()`* the following:

```
onMouseUp => sub {

        my ($button, $btn_code, $x, $y, $mod) = @_;
        return unless $btn_code == mb::Right;

        my $owner = $button->owner;

        # Reset background to white on right-click
        $owner->backColor(cl::White);
        $button->visible(0);
        $owner->repaint;
        $button->visible(1);
    },
```

24.5 How I Built a Custom Timer in Prima

Prima comes with a built-in timer class called `Prima::Timer`. It lets you run code over and over at regular time intervals - for example, once every second. The standard way to use it is by assigning a callback to onTick, which runs every time the timer 'ticks.' To create a timer that tracks elapsed time and offers more control (e.g., start, stop, reset), subclassing is the natural next step. By looking at the original `Prima::Timer` code, I learned that it uses two key methods for customization:

- `profile_default`: sets up default settings (like `timeout => 1000` for 1-second intervals).
- `init`: sets up the actual behavior when the timer is created.

I also saw how Prima uses a `notification_types` hash to define events like `Tick`, and how these events can be triggered with `$self->notify('Tick').`

24.5.1 What is a notification in Prima?

A notification is an event that a widget or object can emit. It's similar to a signal or callback. You can assign a handler for it when you create the object (like `onClick`, `onPaint`, etc.).

This structure made it very inviting to build my own version on top of it.

In `Prima::Timer`, the notification is:

`Tick => nt::Default`

This tells Prima:

"This object can emit a Tick event, and it uses the default type of notification."

So when the timer interval passes, it internally calls:

`$self->notify('Tick');`

This triggers your assigned handler. Where does this come from?

`Prima::Component` provides:

```
sub notify {
    my ($self, $notification, @args) = @_;
    my $handler = $self->can("on$notification");
    $self->$handler(@args) if $handler;
}
```

So when the timer fires, it says:

`$self->notify('Tick');`

Which then calls:

```
$self->onTick();
```

If you assigned an `onTick` handler, it runs.

24.5.2 Creating My Subclass

I created a new package called `My::ElapsedTimer`, which inherits from `Prima::Timer`. In `profile_default`, I added a new field: `elapsed`, starting at 0. In `init`, I added behavior so that every time the timer ticks, it increases `elapsed` by 1 and sends out a custom event I called `Elapsed`.

I also added some convenience methods:

- `start_timer`: resets the counter and starts the timer
- `stop_timer`: stops the timer
- `reset_timer`: sets the counter back to 0
- `elapsed`: returns how many ticks have passed

Now, this custom timer is more than just a tick generator - it's a mini time tracker I can drop into any Prima app.

24.5.3 Why This Helped Me Learn

By building my own subclass, I didn't just write new code - I learned how Prima is structured. It showed me how components in Prima are designed to be extended: clean separation of defaults, initialization, and behavior. I also got hands-on practice with how event notifications work under the hood.

And the best part? My new timer works just like a built-in widget, but it does exactly what I need.

24.5.4 My Custom Timer

First an overview of the implemented methods and their purpose.

Method	Purpose
`profile_default`	Sets initial `timeout`, `elapsed`
`init`	Hooks up the `onTick` to update `elapsed` and send `Elapsed`
`elapsed`	Returns how many ticks have passed
`reset_timer`	Resets counter to 0
`start_timer`	Resets and starts
`stop_timer`	Stops the timer

Table 24.1: Implemented Methods in the Custom Timer and Their Purposes

So let's have a look at the code with some annotations. First the package definition:

```
package My::ElapsedTimer;
use base 'Prima::Timer';
# Add a new notification type if needed (optional)
my %RNT = (
    %{Prima::Timer->notification_types},
    Elapsed => nt::Default,
);
sub notification_types { return \%RNT; }
# Set default profile values
sub profile_default {
    my $def = $_[0]->SUPER::profile_default;
    $def->{timeout} = 1000;     # tick every 1 second
    $def->{elapsed} = 0;        # track elapsed time
    return $def;
}
# Initialization code
sub init {
    my ($self, %profile) = @_;
    $self->{elapsed} = 0;
    # Set up the ticking behavior
    $self->onTick(sub {
        $self->{elapsed}++;
        $self->notify('Elapsed');  # emit custom Elapsed event
    });
    return $self→SUPER::init(%profile);
}

# Public method to access elapsed time
sub elapsed { $_[0]->{elapsed} }

# Reset timer count
sub reset_timer {
    my $self = shift;
    $self->{elapsed} = 0;
}
# Convenience methods
sub start_timer {
    my $self = shift;
    $self->reset_timer;
    $self->start;
}
sub stop_timer {
    my $self = shift;
    $self->stop;
}
1;
```

Now use the subclass in your main script:

```
use Prima qw(Application Buttons Label);
use lib '.'; # or adjust to path where My/ElapsedTimer.pm is
use My::ElapsedTimer;

my $mw = Prima::MainWindow->new(
    text => 'Custom Timer Demo',
    size => [220, 150],
    icon => Prima::Icon->load('icon.png'),
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
);
my $label = $mw->insert('Prima::Label',
    text   => 'Elapsed: 0',
    origin => [20, 80],
    size   => [200, 40],
    font   => { size => 16 },
);
my $button = $mw->insert('Prima::Button',
    text   => 'Start',
    origin => [20, 30],
    size   => [80, 30],
);
my $reset_button = $mw->insert('Prima::Button',
    text   => 'Reset',
    origin => [120, 30],
    size   => [80, 30],
);
my $timer;
$timer = My::ElapsedTimer->new(
    timeout => 1000,
    enabled => 0,
    onElapsed => sub {
        $label->text("Elapsed: " . $timer->elapsed);
    },
);
my $running = 0;
$button->onClick(sub {
    if ($running) {
        $timer->stop_timer;
        $button->text('Start');
        $running = 0;
    } else {
        $timer->start_timer;
        $button->text('Stop');
        $running = 1;
    }
});

$reset_button->onClick(sub {
    $timer->reset_timer;
    $label->text('Elapsed: 0');
});

Prima->run;
```

Listing 24.2: Package main My Custom Timer

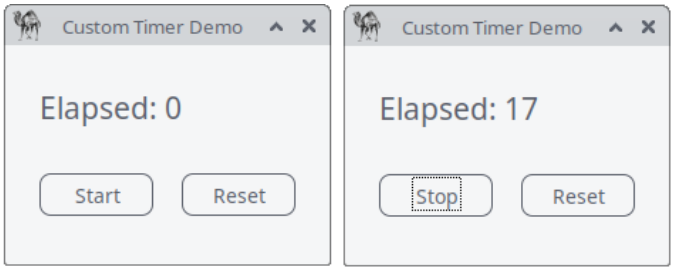


Figure 24.5: Custom timer in action

Closing Words and an Exercise!

One of the most powerful lessons I've learned - and continue to learn - is that **true mastery comes from doing**. Writing this guide, experimenting with Prima's Visual Builder, and building custom components like the `My::ElapsedTimer` have reinforced this idea time and again. Every day, I discover something new, not just about Prima, but about problem-solving, design, and the art of coding itself.

Why "Learning by Doing" Works:

- **Hands-On Exploration**: The Visual Builder isn't just a tool for creating UIs; it's a playground for curiosity. By dragging, dropping, tweaking, and sometimes breaking things, I've uncovered nuances in Prima's architecture that no documentation could fully capture.
- **Iterative Improvement**: The process of refining code—like fixing the repaint artifacts or adding right-click functionality—teaches resilience and adaptability. Each challenge is an opportunity to grow.
- **Building Intuition**: The more I experiment, the more intuitive Prima's patterns become. What once seemed complex (like subclassing or event notifications) now feels like second nature.

A Daily Practice: I encourage you to adopt this mindset: **spend a little time each day experimenting**. Try modifying an existing example, building a small tool, or even just exploring the Object Inspector. Over time, these small efforts compound into deep understanding and confidence.

Final Encouragement: The journey of learning never truly ends—it evolves. Whether you're a beginner or an experienced developer, there's always something new to discover. Embrace the process, celebrate the small wins, and remember: every line of code you write is a step forward.

Now, go build something amazing!

Looking for a project idea? Build your own mini task-manager app!

1. Overview

The **Mini Task Manager App** is a simple yet functional application built using the **Prima toolkit** and its **Visual Builder**. This app allows users to manage tasks, track time spent on each task, and perform basic operations like adding, completing, and deleting tasks. It serves as a practical exercise to reinforce concepts such as UI design, event handling, and custom component integration.

2. Objectives

- Design a user interface using Prima's Visual Builder.
- Implement event handlers for user interactions.
- Use a custom timer component to track task duration.
- Practice UI updates and repainting techniques.
- Create a functional, user-friendly application.

3. Functional Requirements

3.1. User Interface

- **Main Window**: A window titled "Mini Task Manager" with a size of 400x300 pixels.
- **Task List**: A `Prima::ListBox` widget to display tasks.
- **Buttons**:

    - **Add Task**: A button to add a new task.
    - **Complete Task**: A button to mark the selected task as complete.
    - **Delete Task**: A button to remove the selected task.
- **Timer Label**: A `Prima::Label` widget to display the elapsed time for the selected task.

3.2. Task Management

- **Add Task**:
    - Prompt the user to enter a task name using a dialog box.
    - Add the task to the task list.
- **Complete Task**:
    - Mark the selected task as complete.
    - Stop the timer for the task.
- **Delete Task**:
    - Remove the selected task from the task list.
    - Stop the timer for the task.

3.3. Timer Functionality

- **Start Timer**: Start the timer when a task is selected.
- **Stop Timer**: Stop the timer when a task is completed or deleted.
- **Display Elapsed Time**: Update the timer label to show the elapsed time in seconds.

4. Non-Functional Requirements

4.1. Usability

- The UI should be intuitive and easy to navigate.
- Buttons should be clearly labeled.
- The task list should be easy to read and interact with.

4.2. Performance

- The app should respond quickly to user interactions.
- UI updates should be smooth and free of visual artifacts.

4.3. Code Quality

- The code should be well-structured and modular.
- Use comments to explain key sections of the code.
- Follow best practices for event handling and UI updates.

5. Technical Specifications

5.1. Tools and Libraries

- **Prima Toolkit**: Use Prima's Visual Builder for UI design.
- **Custom Timer**: Use the `My::ElapsedTimer` class for tracking task duration.

5.2. Code Structure

- **Main Window Class**: `TaskManagerWindow` (subclass of `Prima::MainWindow`).

- **Methods**:
  - `profile_default`: Set default properties for the window.
  - `init`: Initialize UI components and event handlers.
  - `add_task`: Add a new task to the list.
  - `complete_task`: Mark the selected task as complete.
  - `delete_task`: Remove the selected task.
  - `select_task`: Start the timer for the selected task.
  - `stop_timer`: Stop the timer and reset the timer label.

5.3. Event Handlers

- **Button Clicks**: Handle clicks for "Add Task," "Complete Task," and "Delete Task."
- **Task Selection**: Start the timer when a task is selected.

6. Deliverables

6.1. Source Code

- A Perl script (`task_manager.pl`) containing the complete implementation of the Mini Task Manager App.

6.2. Documentation

- A brief document explaining the app's functionality, code structure, and how to run it.

6.3. Screenshots

- Screenshots of the app in action, showing the UI and timer functionality.

7. Evaluation Criteria

7.1. Functionality

- Does the app allow users to add, complete, and delete tasks?
- Does the timer correctly track and display elapsed time?

7.2. Usability

- Is the UI intuitive and easy to use?
- Are there any visual or functional issues?

7.3. Code Quality

- Is the code well-structured and modular?
- Are there comments explaining key sections?

7.4. Creativity

- Are there any additional features or improvements beyond the basic requirements?

8. Instructions for Students

1. **Design the UI**: Use Prima's Visual Builder to create the main window and add the required widgets.
2. **Implement Functionality**: Write the code to handle task management and timer functionality.
3. **Test the App**: Run the app and test all features to ensure they work as expected.
4. **Refine and Extend**: Improve the UI and add any additional features you think would be useful.
5. **Submit Your Work**: Provide the source code, documentation, and screenshots.

9. Conclusion

This exercise is designed to help you apply what you've learned about Prima's Visual Builder, event handling, and custom components. By building the Mini Task Manager App, you'll gain practical experience in creating functional and user-friendly applications. Good luck, and have fun coding!

Part 11 – Miscellaneous Topics

These chapters gather a collection of useful Prima features that don't quite fit into the main flow of the tutorial, yet are incredibly handy for beginners and intermediate users alike. They showcase practical, fun techniques that open up new possibilities and help your applications do even more. Think of this part as your "Prima toolbox": small, focused skills you can plug into any project.

25. Markup in Widgets

Markup makes your UI richer: colored labels, styled text, icons inside text widgets. You will love this because it immediately improves appearance (as you already saw). Markup works in all text-bearing widgets, including:

- `Label`
- `Button`
- `Radio` / `CheckBox`
- `GroupBox`
- `ListBox`
- `DetailedList`
- `StringOutline`
- Custom canvases via `text_out`

`markup.pl`, located in the example directory of the Prima package, is an outstanding and canonical demonstration of:

- basic markup syntax
- nested formatting (B<>, I<>, U<>, C<color|text>, etc.)
- fonts via `F<>`
- superscript/subscript (S<>)
- placement and alignment (P<> and M<>)
- embedded images via IMAGES array + `I<>`
- usage inside widgets (Label, Button, ListBox, DetailedList, Outline, Canvas)
- BiDi support and rotated text
- tooltips and links (Lpod://…|text)

Tag	Usage	Example
`B<>`	Bold	`B`
`I<>`	Italic	`I`
`U<>`	Underline	`U`
`C | text>`	Colored text	
`S<+2 | text>`	Increase font size	
`S<-2 | text>`	Decrease font size	
`W`	Non-wrappable text	`W`
`F | text>`	Font selection by ID	

Tag	Usage	Example
`P`	Paragraph alignment	`P<0>Left aligned text`
`L | text>`	Tooltip or link	
`M`	Move text position	`M<5,0,Shifted right>`
`G | text>`	Background color	
`I`	Embedded image by ID	`I<1>`
`S< | text>`	Superscript	
`S | >`	Subscript	

Table 25.1 Markup tag reference table

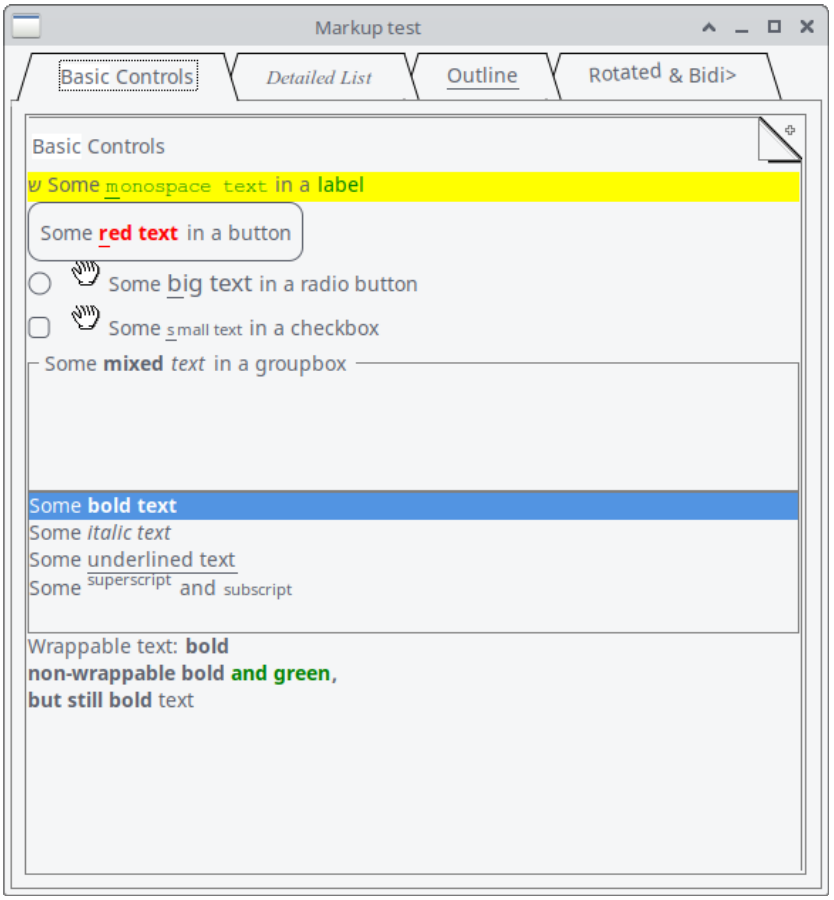


Figure 25.1 Basic Controls Markup Application

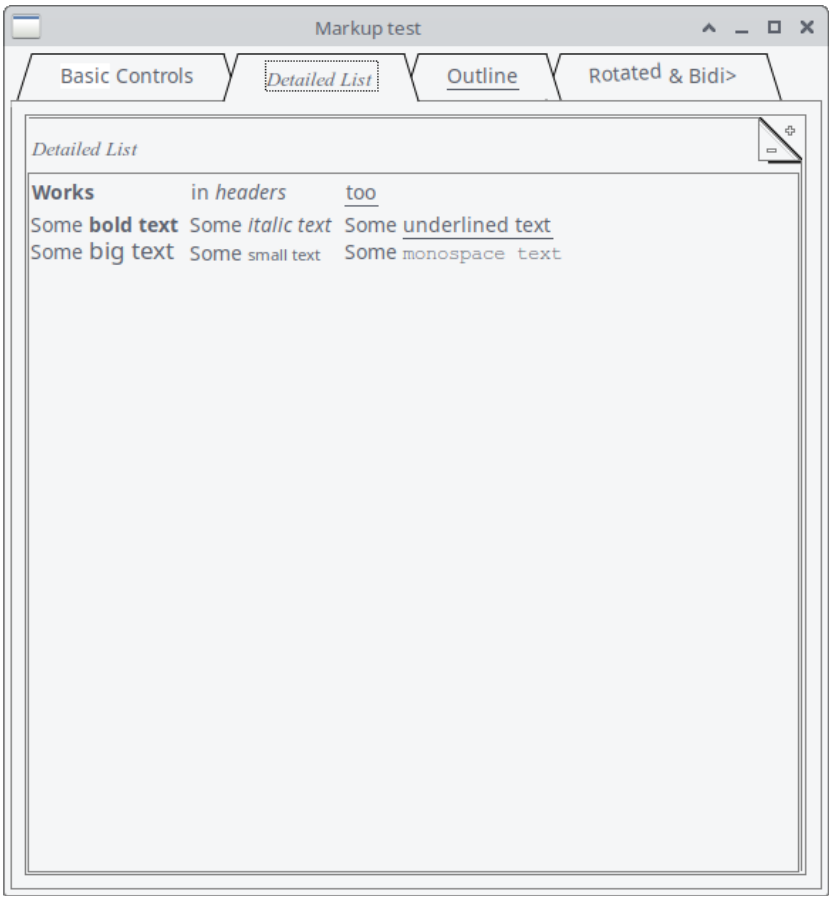


Figure 25.2 Detailed List Markup Application

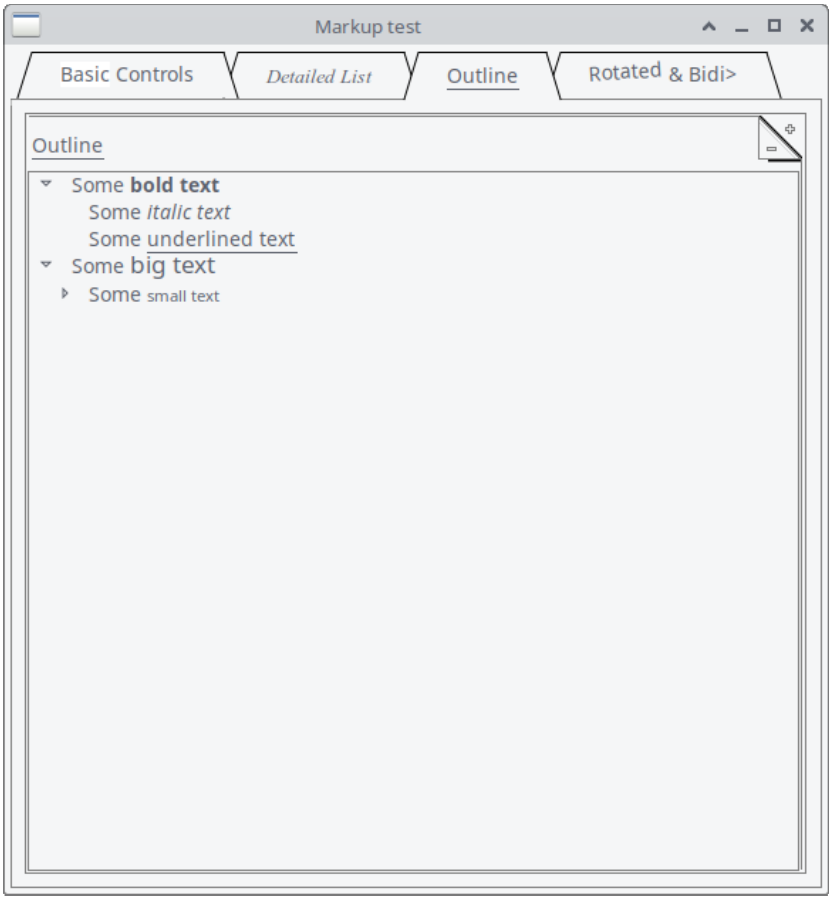


Figure 25.3 Outline Markup Application

Figure 25.4 Rotated and bidi Markup Application

The following example brings all markup features together in one application.

```
# Load Prima modules for GUI widgets
use Prima qw(Application Buttons Edit Notebooks Label DetailedList Outlines MsgBox);
use FindBin qw($Bin);

# Load Prima::Drawable::Markup, 'M' function helps convert strings to
# markup objects
use Prima::Drawable::Markup qw(M);

# Structure of this listing
# ----------------------------
# 1. Define available fonts
# ----------------------------
# Each hash defines a font; direction can control RTL text
@Prima::Drawable::Markup::FONTS = (
    { name => 'Times New Roman' },     # index 0
    { name => 'Courier New', direction => 4 }, # index 1
    { name => 'Arial' }, # index 2 -> fixes F<2|Rotated>
);

# ----------------------------
# 2. Define images for markup
# ----------------------------
# Images can be embedded inline with I<id> in markup
@Prima::Drawable::Markup::IMAGES = (
    # Load a GIF from the script directory
    Prima::Icon->load("$Bin/Hand.gif")
);

# ----------------------------
# 3. Create main window
# ----------------------------
my $Main = Prima::MainWindow->new(
    name   => 'Main',
    text   => 'Markup test',
    size   => [500, 500],
    designScale => [7, 16],  # Controls DPI scaling
);

# ----------------------------
# 4. Add a TabbedNotebook widget
# ----------------------------
# Tabs demonstrate different markup and widgets
my $tn = $Main->insert('TabbedNotebook',
    pack   => { expand => 1, fill => 'both' },
    tabs   => [
        M 'G<White|Basic> Controls',           # Tab 0
        M 'F<0|I<Detailed List>>',             # Tab 1
        M 'U<Outline>',                        # Tab 2
        M 'F<2|Rotated> & Bidi>'               # Tab 3
    ],
);

# ----------------------------
# 5. Tab 0: Label with tooltip and monospace
# ----------------------------
$tn->insert_to_page(0, 'Label',
    text      => \ "\x{5e9} Some L<tip://$0/tip|F<1|U<m>onospace text>> " .
                   "in a L<pod://Prima::Label/SYNOPSIS|label>",
    autoHeight => 1,
    hotKey    => 'm',
    backColor => cl::Yellow.
```

```
    wordWrap  => 1,
    focusLink => 'List',
    pack      => { side => 'top', fill => 'x', anchor => 'w' },
);
# Explanation:
# \x{5e9} = Hebrew letter
# L<tip://...|...> = clickable tooltip link
# F<1|U<m>onospace text> = monospace font + underlined letter

# ----------------------------
# 6. Tab 0: Button with markup and click action
# ----------------------------
$tn->insert_to_page(0, 'Button',
    text   => \ 'Some B<C<LightRed|U<r>ed text>> in a button',
    pack   => { side => 'top', anchor => 'w' },
    hotKey => 'r',
    onClick => sub {
        message(\ "Hello! This is the B<msgbox> speaking!")
    },
    hint   => \ "Hints can I<also> be markupified",
);
# Nested tags: Bold -> Colored -> Underlined -> Letter

# ----------------------------
# 7. Tab 0: Radio button with big text
# ----------------------------
$tn->insert_to_page(0, 'Radio',
    text => \ 'P<0>Some S<+2|U<b>ig text> in a radio button',
    pack => { side => 'top', anchor => 'w' },
    hotKey => 'b',
);
# S<+2|…> increases font size, U<b> underlines 'b'

# ----------------------------
# 8. Tab 0: CheckBox with small text
# ----------------------------
$tn->insert_to_page(0, 'CheckBox',
    text => \ 'P<0>Some S<-2|U<s>mall text> in a checkbox',
    pack => { side => 'top', anchor => 'w' },
    hotKey => 's',
);
# S<-2|…> decreases font size, U<s> underlines 's'

# ----------------------------
# 9. Tab 0: GroupBox with mixed text
# ----------------------------
$tn->insert_to_page(0, 'GroupBox',
    text => \ 'Some B<mixed> I<text> in a groupbox',
    pack => { side => 'top', fill => 'x' },
);
# B<> = bold, I<> = italic

# ----------------------------
# 10. Tab 0: ListBox with various markup
# ----------------------------
$tn->insert_to_page(0, 'ListBox',
    name        => 'List',
    focusedItem => 0,
    items => [
        M 'Some B<bold text>',
        M 'Some I<italic text>',
        M 'Some U<underlined text>',
```

```
        M 'Some U<underlined text>',
        M 'Some S<+2|big text> M<,-0.4,m> and S<-2|subscript>',
    ],
    pack => { side => 'top', fill => 'x' },
);
# Demonstrates bold, italic, underline, big/small text, superscript/subscript

# ----------------------------
# 11. Tab 0: Wrappable label with nested markup
# ----------------------------
$tn->insert_to_page(0,'Label',
    wordWrap => 1,
    pack => { side => 'top', fill => 'both', expand => 1 },
)->text( \ "Wrappable text: B<bold
W<non-wrappable bold C<Green|and green>>,
but still bold> text"
);
# W<> prevents line wrapping, C<Green|…> colors text

# ----------------------------
# 12. Tab 1: DetailedList with headers and markup
# ----------------------------
$tn->insert_to_page(1,'DetailedList',
    headers => [ M 'B<Works>', M 'in I<headers>', M 'U<too>'],
    items   => [
        [ M 'Some B<bold text>',
          M 'Some I<italic text>',
          M 'Some U<underlined text>'
        ],
        [ M 'Some S<+2|big text>',
          M 'Some S<-2|small text>',
          M 'Some F<1|monospace text>'
        ],
    ],
    columns => 3,
    pack    => { expand => 1, fill => 'both' },
);
# Demonstrates markup in headers and table cells

# ----------------------------
# 13. Tab 2: StringOutline with nested markup
# ----------------------------
$tn->insert_to_page(2,'StringOutline',
    items => [
        [ M 'Some B<bold text>', [
            [ M 'Some I<italic text>'],
            [ M 'Some U<underlined text>'],
        ]],
        [ M 'Some S<+2|big text>', [
            [ M 'Some S<-2|small text>', [
                [ M 'Some F<1|monospace text>' ],
            ]],
        ]],
    ],
    pack => { expand => 1, fill => 'both' },
);
# Nested outline items with bold, italic, underline, size, and monospace

# ----------------------------
# 14. Tab 3: Custom widget with canvas painting
# ----------------------------
$tn->insert_to_page(3,'Widget',
    font => { size => 16, direction => 20, name => 'Arial' },
```

```
    font => { size => 16, direction => 30, name => 'Arial' },
    pack => { expand => 1, fill => 'both' },
    text => \ "G<Yellow|B<I<\x{5E9}\x{5DC}\x{5D5}\x{5DD}> C<Green|world>>!>",
    onPaint => sub {
        my ($self, $canvas) = @_;
        $canvas->clear;
        my ($ox, $oy) = (20, 20);
        $canvas->text_out($self->text, $ox, $oy);
        $canvas->color(cl::LightRed);
        $canvas->fill_ellipse($ox, $oy, 5, 5);
    },
);
# Demonstrates direct canvas painting with markup
# Nested tags: Yellow background, bold + italic Hebrew letters, green word

# ----------------------------
# 15. Run the Prima application
# ----------------------------
Prima->run;

=pod

=head1 tip

This is a tooltip!

=cut
```

Listing 25.1: Markup Application

26. Clipboard

Use *`Prima::Clipboard`* for copying data (text, images, and custom formats) to the clipboard. Think of data exchange between applications, e.g. converting some text or copy and paste images in a document.

26.1 Copying Text to the Clipboard

This next example is a very easy one to demonstrate how the class works.

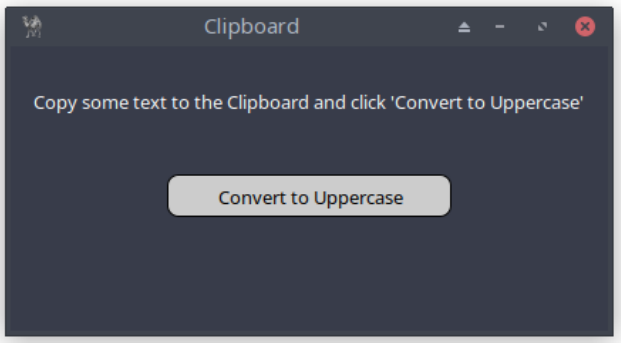


Figure 26.1: Clipboard Application

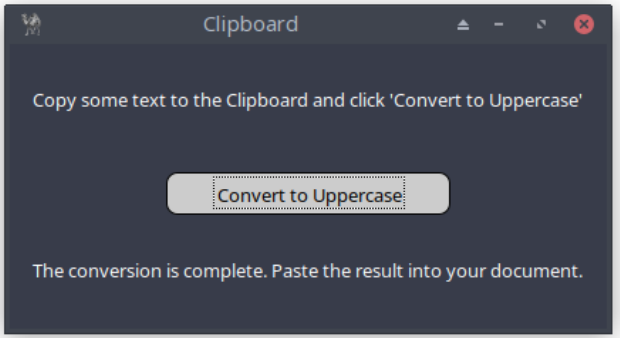


Figure 26.2: Copying Text to the Clipboard

This minimal program shows the absolute basics of reading and writing text to the clipboard.

```
use Prima qw(Label Buttons Application);

my $mw = Prima::MainWindow->new(
    text => 'Clipboard',
    size => [420, 200],
    backColor => cl::Gray,
    font => { size => 10, },
    icon => Prima::Icon->load('icon.png'),
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
);

my $intro_label = $mw->insert( Label =>
    origin => [0,110],
    size => [$mw->width, 60],
    text => "Copy some text to the Clipboard and click " .
            "'Convert to Uppercase'",
    alignment   => ta::Center,
    color => cl::White,
    backColor => cl::Gray,
    ownerFont => 1,
);


my $output_label;
my $success = "The conversion is complete. Paste the result into " .
              "your document.";
my $error = "Nothing to convert...";

my $convert_button = $mw->insert( Button =>
    origin => [($mw->width/2)-(200/2), 80],
    size      => [200, 30],
    text      => "Convert to Uppercase",
    ownerFont => 1,
    color     => 0x000000,
    backColor => 0xcccccc,
    onClick   => sub {
                            $output_label->visible(0);
                            my $c = $::application->Clipboard;
                            my $str = $c->text;
                            if ($str =~ /[a-z]/) {
                               $c->text(uc($str));
                               $output_label->text($success);

                            } else {
                                $output_label->text($error);
                            }
                            $output_label->visible(1);
                        },
);

$output_label = $mw->insert( Label =>
    origin => [0, 0],
    size => [$mw->width, 50],
    text => '',
    alignment => ta::Center,
    wordWrap => 1,
    color => cl::White,
    visible => 0,

    ownerFont => 1,
);
```

```
);

Prima->run;
```

Listing 26.1: Copying Text to the Clipboard

26.2 Copying Images to the Clipboard

In the following program, we use the `Prima::Dialog::FileDialog` module to copy an image.

The `FileDialog`interface enables users to browse, select, and manage files or directories through an intuitive dialog window. One of its most powerful features is the ability to filter files by extension or custom criteria, making it easy to find exactly the files you need.

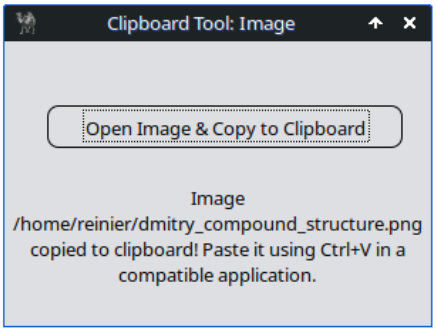


Figure 26.3: Copying Images to the Clipboard

```
use Prima qw(Application Dialog::FileDialog);

my $c = $::application->Clipboard;

my $mw = Prima::MainWindow->new(
    text     => 'Clipboard Tool: Image',
    size     => [300, 200],
    font => { size => 10, },
    icon => Prima::Icon->load('icon.png'),
    borderStyle => bs::Dialog,
    borderIcons => bi::SystemMenu|bi::TitleBar,
);

my $result_label = $mw->insert( Label =>
    origin => [5, 25],
    size   => [290, 60],
    text   => "",
    alignment => ta::Center,
    autoHeight => 1,
    wordWrap => 1,
    ownerFont => 1,
);


$mw->insert(Button =>
    origin => [30, 125],
    size   => [250, 30],
    text    => 'Open Image & Copy to Clipboard',
    ownerFont => 1,
    onClick => sub {
        $result_label->text("");

        my $file_dialog = Prima::Dialog::OpenDialog->new(
            filter => [['Image files' => '*.bmp;*.gif;*.jpg;*.jpeg;*.png']],
            multiSelect => 0,
            # default value is 1, only for demonstration purpose
            sorted => 1,
        );

        if ($file_dialog->execute) {
            my $image_path = $file_dialog->fileName;
            my $image = Prima::Image->load($image_path);

            if ($image) {
                # Copy image to clipboard
                my $img = $c->image( $image );
                $result_label->text("Image $image_path copied to " .
                                    "clipboard! Paste it using Ctrl+V " .
                                    "in a compatible application.");
            } else {
                $result_label->text("Failed to load the image.");
            }
        }
    },
);

Prima->run;
```

Listing 26.2: Copying Images to the Clipboard

Interesting feature is creating Help button in the FileDialog

```
my $file_dialog = Prima::Dialog::OpenDialog->new(
    filter => [['Image files' => '*.bmp;*.gif;*.jpg;*.jpeg;*.png']],
    multiSelect => 0,
    showHelp => 1,
    helpContext => "pod.pl",
);
```

wherewhere pod.pl is defined as:

```
=pod

=head1 HELP

Select a file or type a filename and click the button 'Open'.

=cut
```

26.3 Notes on Custom Clipboard Formats

Besides text and images, the clipboard in Prima can also store custom data formats. These are mainly useful when two Prima applications want to exchange their own internal data, such as settings or structured information. Each custom format is identified by a name and is ignored by other programs that do not understand it.

For most beginners and everyday applications, working with plain text and images is more than enough. Custom formats are simply there as an extra option if, later on, you build more advanced tools that need to copy and paste specialized data.

27. Calendar - A Selectable Date Widget

The *`Prima::Calendar`* widget displays a month view with selectable dates. This example shows how to:

- enable or disable locale support
- respond to date changes
- reset the calendar to today
- change the first day of the week
- attach calendar behavior to menu items

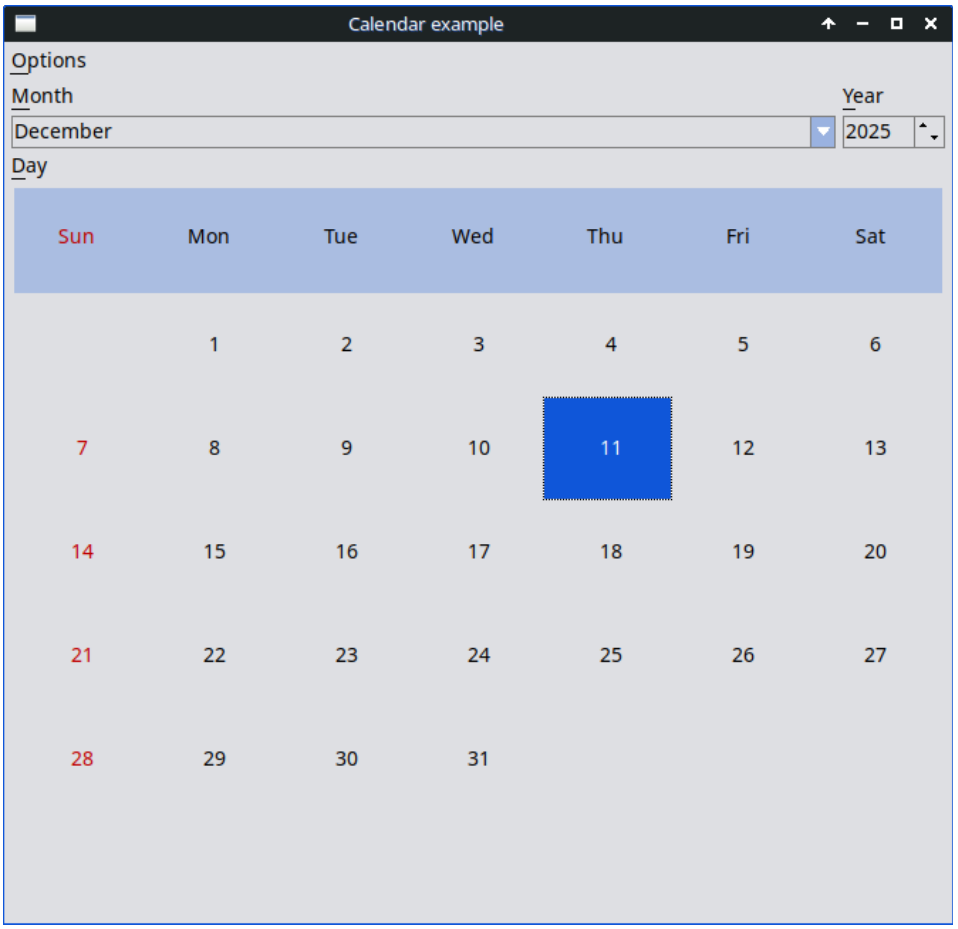


Figure 27.1: Calendar

This widget is ideal for date selection, schedules, reminders, or small tools that need date input.

```
# examples/calendar.pl - Standard calendar widget
#
# Demonstrates usage of Prima::Calendar, menu integration, locale switching,
# and reacting to date selection changes.

use Prima;
use Prima::Application name => 'Calendar';
use Prima::Calendar;

my $cal;   # calendar widget (declared here so menu callbacks can see it)

# Create main window with a menu bar for controlling the calendar.
my $w = Prima::MainWindow->new(
    text        => "Calendar example",
    size        => [ 500, 500 ],
    # scale for fonts/layout (not critical for logic)
    designScale => [6,16],

    # Menu with options affecting the calendar widget.
    menuItems => [[ "~Options" => [
        # Toggle locale-based month/day names.
        [ '@locale', 'Use ~locale', 'Ctrl+L', '^L', sub {
            my ( $self, $mid, $val ) = @_;

            my $newstate;
            # attempt to enable/disable locale
            $cal->useLocale( $newstate = $val );
            # force calendar redraw
            $cal->notify(q(Change));

            # If enabling locale failed, uncheck the menu item and
            # warn the user.
            return unless $newstate && !$cal->useLocale;
            $self->menu->uncheck($mid);
            Prima::message("Selecting 'locale' failed");
        }],

        # Reset calendar to today's date.
        [ 'Re~set to current date', 'Ctrl+R', '^R', sub {
            $cal->date_from_time( localtime(time) );
        }],

        # Toggle Monday as the first day of the week.
        [ '@monday', '~Monday is the first day of week', sub {
            my ( $self, $mid, $val ) = @_;
            $cal->firstDayOfWeek($val);
        }],
    ]]],
);

# Insert the calendar widget into the main window.
$cal = $w->insert( Calendar =>
    useLocale => 1,   # try using locale formatting at startup

    # When the selected date changes, update the window title.
    onChange => sub {
        $w->text( "Calendar - " . $cal->date_as_string );
    },

    pack => { expand => 1, fill => 'both' },  # fill window space
);
```

```
# If locale support is active, reflect it in the menu.
$w->menu->locale->check if $cal->useLocale;

Prima->run;
```

Listing 27.1: Calendar

28. Spinner - A Simple Animated Widget

The `Prima::Spinner` widget is a compact animated indicator - perfect for showing background activity or progress. This example demonstrates three spinner styles, a button that starts and stops them, and a slider that controls the spinner's animation phase.

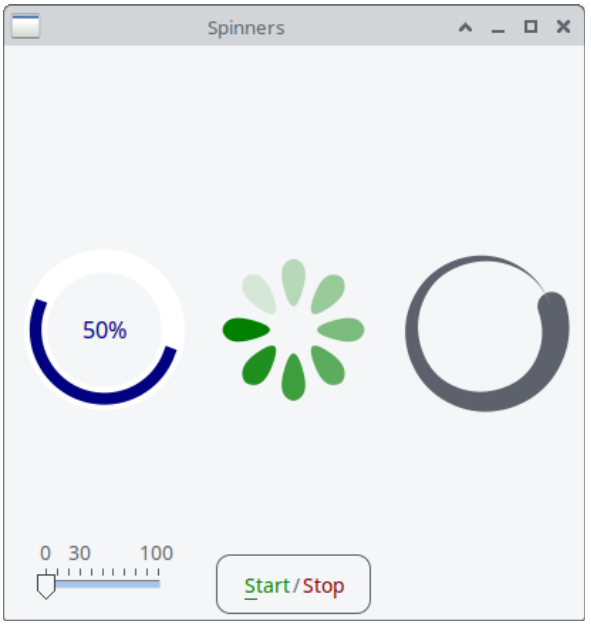


Figure 28.1 Spinner

Spinners don't track real time: the animation is purely visual. You control the speed and phase manually.

```
# examples/spinner.pl - standard spinner widget
#
# Demonstrates several spinner styles and basic interaction:
# - starting/stopping animation
# - changing spinner value
# - embedding a slider inside a spinner

use Prima qw(Application Buttons Spinner Sliders);

# Create the main application window.
my $mw = Prima::MainWindow->new(
    size => [400, 400],
    text => 'Spinners'
);

# First spinner: circular style, custom colors.
my $spinner = $mw->insert('Spinner',
    style       => 'circle',    # choose animation style
    color       => cl::Blue,    # main color
    hiliteColor => cl::White,   # highlight color
    pack        => { side => 'left', fill => 'both', expand => 1 },
);

# Second spinner: "drops" animation style.
my $spinner2 = $mw->insert('Spinner',
    style => 'drops',
    color => cl::Green,
    pack  => { side => 'left', fill => 'both', expand => 1 },
);

# Third spinner: "spiral" style, default colors.
my $spinner3 = $mw->insert('Spinner',
    style => 'spiral',
    pack  => { side => 'left', fill => 'both', expand => 1 },
);

# Button that toggles (starts/stops) ALL spinners.
$mw->insert(
    'Button',
    text     => \ 'C<Green|U<S>tart>/C<Red|Stop>',
    hotKey   => 's',
    checkable => 1, # works like an ON/OFF switch
    checked   => 0, # start in "off" state
    origin    => [0,0],
    onClick   => sub {
        $_->toggle for $spinner, $spinner2, $spinner3;  # toggle animations
    },
    growMode  => gm::XCenter,
);

# Slider INSIDE the first spinner: controls its animation position.
$spinner->insert(
    'Slider',
    min     => 0,
    max     => 100,                      # full animation cycle
    current => 1,                        # starting value
    origin  => [0,0],
    onChange => sub {
        $spinner->value( shift->value );  # update spinner phase
    },
    growMode => gm::XCenter,
```

```
);

Prima->run;
```

Listing 28.1: Spinner

Closing Words

In this final part, you explored a collection of smaller yet powerful features that broaden what your applications can do. Markup gives your widgets expressive formatting, the clipboard enables smooth interaction with the system, and the calendar and spinner widgets offer convenient ways to select dates or animate small interface elements.

These topics may seem diverse, but together they highlight the flexibility and reach of the Prima toolkit. They let you refine your applications, polish the user experience, and add the finishing touches that make software feel complete.

With this, you've reached the end of the book's journey. You now have the knowledge to build interfaces that are practical, dynamic, and thoughtfully designed. The next step is yours: experiment, create, and shape Prima into the applications you imagine.

And browse the full Prima examples folder - it's a treasure chest of ideas. There is much more to discover!

Appendix A: Naming Conventions

While abbreviations like **btn** for "Button" are common, many developers opt for full words to ensure clarity and maintainability. Whether you choose abbreviations or full names, the key is consistency across your project. If you use **btn** for one button, use **btn** for all buttons rather than mixing in *button* or *but*.

The following list provides a reference for abbreviations and full names.

Abbreviation	**Widget / Class Name**
btn	Button
chk	CheckBox
combo	ComboBox
grp	GroupBox
input	InputLine
label	Label
list	ListBox
radio	RadioButton
radiolist	RadioButtonList
scrollbar	Scrollbar
img	Image
imgbtn	ImageButton
link	Hyperlink
form	Form
frame	Frame
frameset	Frameset

Abbreviation	Widget / Class Name
dlg	Dialog
edit	Edit
grid	Grid
nb	TabbedNotebook
obj	Object
timer	Timer
spinbtn	SpinButton
spinedit	SpinEdit
widget	Widget
win	Window

You can make a name more descriptive by adding a suffix (or a prefix). A few examples:

- btnCancel
- chkPerl
- comboColor
- frameSidebar
- imgLogo
- inputEmail
- ok_btn
- translate_btn
- etc.

Appendix B: Installation Prima

from README.md in Prima package

Debian/Ubuntu

```
apt-get install libgtk-3-dev libgif-dev libjpeg-dev libtiff-dev libxpm-dev \
libwebp-dev libfribidi-dev libharfbuzz-dev libthai-dev libheif-dev libfreetype-dev \
libjxl-dev
```

FreeBSD

```
pkg install gtk3 fribidi harfbuzz libxpm libthai pkgconf tiff webp \
giflib freetype2 libheif Xrandr libXcomposite libXcursor \
libXft fontconfig
```

NetBSD

```
pkg install harfbuzz freetype2 png libthai jpeg tiff libheif libXft libXcursor
giflib pkg-config fribidi libjxl libwebp
```

OpenSUSE

```
zypper install gtk3-devel giflib-devel libjpeg-devel libtiff-devel
libXpm-devel libXrandr-devel libXcomposite-devel libXcursor-devel
libfribidi-devel libwebp-devel libharfbuzz-devel libthai-devel
libheif-devel libfreetype-devel libjxl-devel
```

Solaris

Download and install Oracle Developer Studio as the vendor-provided perl is compiled with `cc`, not `gcc`.

Cygwin

Install apt-cyg:

```
wget https://raw.githubusercontent.com/transcode-open/apt-cyg/master/apt-cyg -O /usr/bin/apt-cyg
chmod +x /usr/bin/apt-cyg
```

Install prerequisites:

```
apt-cyg install libgtk3.0-devel libfribidi-devel libgif-devel libjpeg-devel libtiff-devel libXpm-devel
libwebp-devel libharfbuzz-devel libthai-devel libheif-devel libfreetype-dev
```

Win32-Strawberry

Version 5.32 had the necessary libraries included, but version 5.38 does not. Install the following packages:

- libfribidi-1.0.10-win64.zip
- libthai-0.1.29-win64.zip
- libheif-1.17.6-win64.zip
- libwebp-1.0.2-win64.zip
- libjxl-0.11.1-win64.zip

Win32-ActiveState

ActiveState currently doesn't support local compilations. The official answer is here:

https://community.activestate.com/t/how-to-install-gcc-and-or-mingw/10993/2

Consider migrating to Strawberry Perl.

MacOSX

You'll need **homebrew**, **XQuartz**, and a set of extra libraries.

Install homebrew:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Install XQuartz:

```
brew install --cask xquartz
```

Install support libraries:

```
brew install libiconv libxcomposite libxrandr libxcursor libxft fribidi fontconfig
freetype giflib gtk+3 harfbuzz jpeg libpng libtiff webp libxpm libheif jpeg-xl
```

> **Note:** If you plan on using Prima::OpenGL, compile Prima with `WITH_HOMEBREW=0`.

> **Note:** If Prima crashes in `libxft`, remove libxft and install custom-built xorg libraries:

```
brew install dk/x11/xorg-macros dk/x11/libxft
```

or

```
brew tap linuxbrew/xorg
brew install linuxbrew/xorg/libxft
```

Install libthai:

```
brew install dk/libthai/libthai
```

Optional dependencies

Graphic libraries

Prima can use several graphic libraries to handle image files. Compiling Prima with at least one library (preferably for GIF files) is strongly recommended, because internal library images are stored in GIF format.

Supported libraries:

- libXpm
- libpng
- libjpeg
- libgif
- libtiff
- libwebp / libwebpdemux / libwebpmux
- libheif
- libjxl

libheif is not widespread yet. See Prima/Image/heif.pm for details.

Bidirectional input and complex scripts

To support bidirectional Unicode text input and output you need the **fribidi** library.

For Unix builds you will also need **harfbuzz** for complex scripts and font ligatures.

Prima can compile without these libraries, but support will be limited.

Thai text wrapping requires the **libthai** library.

GTK3 / GTK2

Building Prima with GTK3 or GTK2 on X11 installations is recommended, because Prima will then use GTK fonts, colors, and file dialogs.

```
perl Makefile.PL WITH_GTK2=0 WITH_GTK3=0
```

Source distribution installation

Create the Makefile and build:

```
perl Makefile.PL
make
make test
make install
```

You may also run:

```
make xtest
```

If *perl Makefile.PL* fails, the compilation history can be found in makefile.log.

If you see this error:

```
** No image codecs found
```

Then image libraries are missing from your system path.

If libraries or headers are not found, specify paths manually:

```
INC=-I/some/include
LIBS=-L/some/lib
```

Binary distribution installation

Binary distributions are available only for **MSWin32**. Other platforms should install from source.

```
perl ms_install.pl
```

You must manually patch *Prima::Config.pm* if you want to compile modules that depend on Prima.